2009 International Symposium on Code Generation and Optimization

Stream Compilation for Real-time Embedded Multicore Systems

Yoonseo Choi*, Yuan Lin*, Nathan Chong†, Scott Mahlke*, and Trevor Mudge*
*Advanced Computer Architecture Laboratory, University of Michigan, Ann Arbor, MI
{yoonseo, linyz, mahlke, tnm}@umich.edu
†ARM, Ltd. Cambridge, United Kingdom
nathan.chong@arm.com

***Abstract*—Multicore systems have not only become ubiquitous in the desktop and server worlds, but are also becoming the standard in the embedded space. Multicore offers programability and flexibility over traditional ASIC solutions. However, many of the advantages of switching to multicore hinge on the assumption that software development is simpler and less costly than hardware development. However, the design and development of correct, high-performance, multi-threaded programs is a difficult challenge for most programmers. Stream programming is one model that has wide applicability in the multimedia, signal processing, and networking domains. Streaming is convenient for developers because it separates the creation of actors, or functions that operate on packets of data, from the flow of data through the system. However, stream compilers are generally ineffective for embedded systems because they do not handle strict resource or timing constraints. Specifically, real-time deadlines and memory size limitations are not handled by conventional stream partitioning and scheduling techniques. This paper introduces the SPIR compiler that orchestrates the execution of streaming applications with strict memory and timing constraints. Software defined radio or SDR is chosen as the application space to illustrate the effectiveness of the compiler for mapping applications onto the IBM Cell platform.**

***Keywords*-Multicore; streaming applications;**

I. Introduction

Multicore has emerged as the dominant paradigm for high-performance computing. Example systems include the Sun UltraSparc T1 that has 8 cores, the Sony/Toshiba/IBM Cell processor that consists of 9 cores, the NVIDIA GeForce 8800 GTX that contains 16 streaming multiprocessors each with eight processing units, and the Cisco CRS-1 Metro router that utilizes 192 Tensilica processors. Putting more cores on a chip increases peak performance, but assumes the programmer and/or compiler have identified large amounts of thread-level parallelism (TLP) to exploit 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 Java, 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, particularly those used in the embedded space. Stream languages are motivated by the application style used in image processing, graphics, networking, and other media processing domains. Example stream languages are StreamIt [1], Brook [2], CUDA [3], SPUR [4], Cg [5], Baker [6], and Spidle [7]. Stream languages are generally variants of synchronous dataflow wherein the application is represented as a directed graph (stream graph) where each node represents an actor and each arc represents the flow of data [8]. The number of data samples produced and consumed by each node are statically specified. For this work, we utilize the StreamIt model where a program is represented as a set of autonomous actors (called filters) that communicate through first-in first-out (FIFO) data channels [1]. During program execution, actors fire repeatedly in a periodic schedule [9], [10]. 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 capabilities of each processing element, interconnect bandwidth, and memory 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.

For this work, we focus on streaming for embedded multicore systems. Software defined radio (SDR) is selected as the area of focus because of its streaming nature. Traditionally, the physical layer of wireless protocols is implemented with fixed function ASICs. SDR promises to deliver a cost effective and flexible solution by implementing the wide variety of wireless protocols in software. Some

978-0-7695-3576-0/09 $25.00 © 2009 IEEE
DOI 10.1109/CGO.2009.27

IEEE computer society

of the greatest advantages of SDR are based on the “software” aspect of the implementations. SDR promises greater flexibility, multi-mode operation, lower engineering efforts and costs, and shorter time-to-market. These are all based on the assumption that software development is easier than hardware development.

In a wireless protocol, the execution behavior is relatively static. However, real-time latency constraints are required due to the external environment (e.g., synchronization with the bay station). Further, multicore hardware used in SDR systems must be extremely lean to meet tight power constraints of a mobile terminal. In particular, the size of the local memory associated with each core is often small. Caches are typically not used due to their cost and power overhead, thus the burden of memory management is on the compiler. Multiple buffering for reducing data transferring latency complicates this problem. Prior work on stream compilation is focused on more general purpose architectures, such as Raw, and uses relatively simple applications [9], [10]. Thus, they do not consider memory or latency constraints during the partitioning an mapping phases of the compiler.

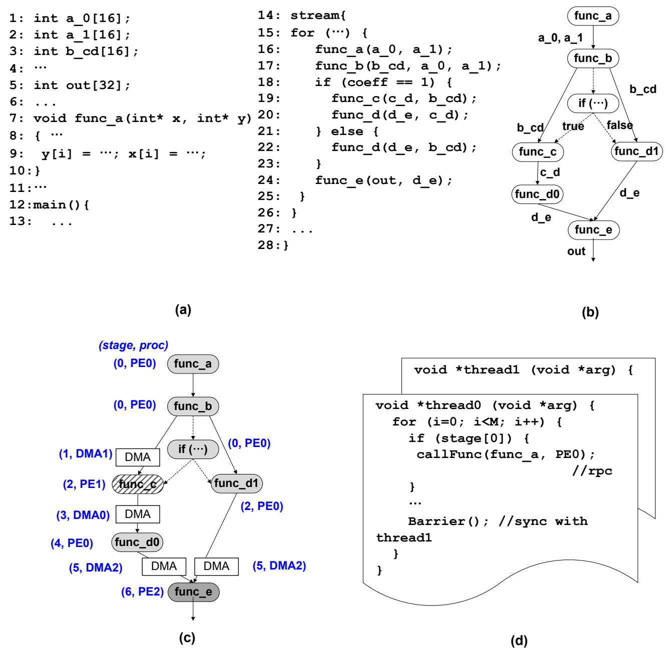
In this paper, we present a coarse-grained software pipelining approach for mapping streaming applications onto embedded multicore systems. This work extends [10] to deal with memory and timing constraints. The output of the scheduler is an assignment of stream kernels to cores and time slots, allocation of variables to local memory, and generation of explicit data transfer operations (DMAs) to move data from core to core. The problem is formulated as an integer linear program and solved using the CPLEX commercial solver. The scheduler is part of a larger compilation system called SPIR (Signal Processing Intermediate Representation). SPIR takes the C language with minimal extensions for representing dataflow programming paradigm as input, rather than the specialized streaming language such as *StreamIt*. Using C can minimize the entry-level efforts for programmers to describing the streaming programming paradigm.

The contributions of this paper are summarized below.

- A stream scheduler that maximizes performance by exploiting pipelining parallelism while satisfying memory constraints imposed by the target hardware.
- A scheduler that provides both the lower and upper bounds on the number of cores which can meet the timing constraints given by the application.
- The effectiveness of SPIR is evaluated using real wireless protocols on the IBM Cell system.
- We compare the effectiveness of our scheduler under memory and latency constraints with [10] to assess its benefits.

II. The Compilation Overview

SPIR is comprised of a frontend translation from the input C program into dataflow graphs; task scheduling; and code generation of threaded code for a multicore target architecture. The overall task-level compilation flow is shown in Figure 1.


```
1: int a_0[16];
2: int a_1[16];
3: int b_cd[16];
4: …
5: int out[32];
6: ...
7: void func_a(int* x, int* y)
8: { …
9:  y[i] = …; x[i] = …;
10:}
11:…
12:main(){
13:   ...
```
```
14: stream{
15: for (…) {
16:    func_a(a_0, a_1);
17:    func_b(b_cd, a_0, a_1);
18:    if (coeff == 1) {
19:      func_c(c_d, b_cd);
20:      func_d(d_e, c_d);
21:    } else {
22:      func_d(d_e, b_cd);
23:    }
24:    func_e(out, d_e);
25:  }
26: }
27: ...
28:}
```

```
void *thread1 (void *arg) {
```
```
void *thread0 (void *arg) {
  for (i=0; i<M; i++) {
    if (stage[0]) {
     callFunc(func_a, PE0);
                        //rpc
    }
    …
    Barrier(); //sync with
thread1
  }
}
```

Figure 1. The SPIR task-level compilation: (a) a sequential input program, (b) a stream graph, (c) a task scheduling, and (d) a parallelized output code

Input language The input language to SPIR is a subset of the C language with several keyword extensions for denoting the streaming scope within an application. Figure 1 (a) illustrates a part of example input. The range enclosed by `stream{ }` is parallelized using coarse-grained software-pipelining. Only one `for` statement is allowed within `stream{ }`, for now. The body of the `for` corresponds to the dataflow or stream graph of the system. In the body of `for` statement, kernel function calls and simple `if` statements are allowed. We currently support one `stream{ }` for a program.

For kernel functions, a set of restrictions is imposed on the conventional C syntax to conform to the dataflow programming model [8]. Streaming data are transferred from one kernel functions to another by passing arguments between them. The arguments of a kernel function should be either read-only or write-only. For example, in Figure 1 (a) at line 16, *func_a* writes only its output data into the arrays *a_0* and *a_1*, which makes *a_0* and *a_1* write-only arguments of *func_a*. These arguments, the data in the arrays, are passed to *func_b* as shown at line 17. *func_b* can read the data in *a_0* and *a_1*, but must not write them. To *func_b*, *a_0* and *a_1* are read-only. Except for the data passed through arguments, all other data used in one kernel should not be shared with any other kernel. Given that all these restrictions are obeyed, SPIR statically infers the type of a function argument such as read-only or write-only. A kernel function can have multiple arguments, where each argument has a fixed size. A kernel cannot handle arguments of changing size. In Figure 1 (a), at lines from 1 to 5, arrays with fixed size are defined. Note that a filter, for example, a FFT (Fast Fourier Transform) can have a token of changing size. To support this, one FFT algorithm can have multiple kernels for different sizes per token. If the size of a token is dependent on run-time environment, `if` statements can be used. Variations in token size can be modestly supported in this fashion.

Based on the information, the frontend of SPIR builds a dataflow or stream graph for the application as in Figure 1 (b). A node corresponds to a kernel function call or to a the condition of an `if` statement. An edge denotes a transfer of data between the two kernel function calls. A control transfer by an `if` statement is also denoted by an edge annotated either with *true* or *false*. In this paper, a node, a kernel function call, and a task are all used interchangeably.

Task scheduling Given a stream graph and a target platform, the task scheduler assigns each node in the stream graph to a processor in the target platform. It also allocates stream buffers, and generates DMAs under given memory and timing constraints. Figure 1 (c) shows the result of this scheduling in parentheses. For a node, a software pipeline stage and a processor are assigned. The details on the task scheduling will be described shortly in Section III.

Target architecture Our target architecture can be any multicore platform that has a control processor and multiple data processors or processing elements (PE), such as IBM Cell and ARM Ardbeg [11][12]. In this paper, we used IBM Cell as our target architecture. It has a PowerPC control processor and a number of SPEs. Each SPE is equipped with a software-managed local memory and a DMA engine, and mainly specialized for heavy-duty data processing. Since these PEs often perform best without the presence of control flows, SPIR adopts function-offload model.

For a SPE, one PPE thread is generated, which spawns kernel function calls and DMA operations to the corresponding SPE. A SPE program handles kernel function calls and DMA operations as dictated by the PPE thread. The program image of a PE contains the local data and codes of kernel functions that are mapped to the PE.

Code generation The code generation phase of SPIR generates a thread for each PE, which is controlled by the control processor as shown in Figure 1 (d). Each thread has a kernel-only software pipelined form. The code generation phase adopts a modified predicate execution to support control flows from `if` statements. The condition in an `if` statement is also multiply buffered for the statements in different software-pipelined stages. The statements that were in the body of an `if` are guarded by these buffered `if` conditions. The code generation phase allocates buffers as needed by the task schedule and predicated decisions.

To summarize, SPIR takes a sequential dataflow program represented textually in C language and generates a multi-threaded parallel program for a multicore system. SPIR is aware of the hardware and software constraints and maximizes the performance within those constraints.

III. Modulo Scheduling with Memory and Real-Time Constraints

A. Processor Assignment for Maximizing the Throughput

In this section, we first review the processor assignment in [10], which aims to maximize the throughput. Consider a stream graph $G = (V,E)$ corresponding to a stream program. A node corresponds to a kernel function call or to a the condition of an `if` statement. An edge denotes a transfer of data between the two kernel function calls. Let $M = |V|$, and P be the number of PEs. For software pipelining, each task should be assigned to exactly one PE. To maximize the throughput of the pipelining, we need to minimize the maximum workload among all PEs.

A 0-1 integer variable a_{ij} is introduced for every task i to denote if i is assigned to a PE j. Equation (1) ensures that a task is assigned to exactly one PE.

$$\sum_{j=1}^{P} a_{ij} = 1 \qquad \text{for all } i = 1, \cdots, M \tag{1}$$

The workload of task i on PE j is $w(i) \cdot a_{ij}$. The total workloads for PE j are $\sum_{i=1}^{M} w(i) \cdot a_{ij}$. The objective is to minimize the maximum of these workloads among all processors. The resulting integer linear program (ILP) has the objective function (3), with constraints (1) and (2) [10]. The variable II is the initiation interval of the modulo scheduling. We call this problem PA-ii standing for Processor Assignment for minimizing the Initiation Interval.

$$\sum_{i=1}^{M} w(i) \cdot a_{ij} \le II \qquad \text{for all } j = 1, \cdots, P \tag{2}$$

$$\text{Minimize} \quad II \tag{3}$$

Problem PA-ii is a version of classic P-ary partition problem, which is NP-hard. Our ILP formulation is based on the assumption that the time spent in DMA within II are shorter than II so that DMA time is effectively hidden being overlapped with kernel task's execution, as in [10]. As shown in Figure 2 (c), a DMA operation executes in parallel with task v.

B. Processor Assignment for Maximizing the Throughput under Memory Constraints

PA-ii assumes infinite local memory per PE. However, a PE has a fixed amount of memory. We extend the processor assignment to be aware of memory constraints. We consider a buffer allocation scheme in which all the buffers used for a task are allocated within the memory of the processor where the task is assigned.

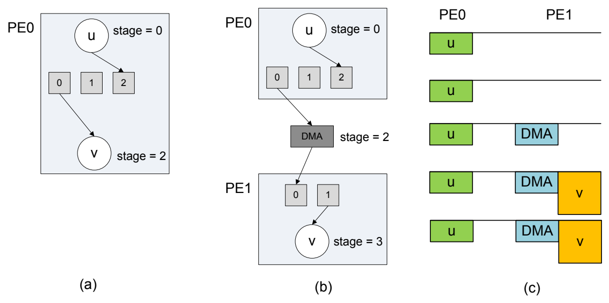


Figure 2. An example of multiple buffering in the software pipelining: (a) when both producer and consumer tasks are in the same processor, and (b) when they are in different processors. Finally, (c) shows the pipelined execution of (b).

In a software pipelined code, multiple buffers are introduced to keep up with the difference in the stages between two tasks. Figure 2 illustrates how many buffers are introduced [10]. Let s_u denote the pipelined stage of node u. When two nodes u and v are in the same processor, the number of buffers for saving an output argument of node u that will be fed to another node v as input is $s_v - s_u + 1$. Those same buffers are also used as input to node v. When two nodes u and v are in two different processors $PE0$ and $PE1$, a DMA operation d that copies the output of u in $PE0$'s memory to the input of v in $PE1$'s memory is needed. The number of buffers for an output of u is now $s_d - s_u + 1$, while that for an input of v is $s_v - s_d + 1$, ($s_u < s_d \le s_v$). The total amount of data buffers a task (i.e. a node) uses is calculated by aggregating the buffer usages of all the output and input arguments of the task. The total amount of buffers due to the output arguments for task u is

$$out\_bufs(u) = \sum_{\forall i \in \{\text{output args of } u\}} \Big(\max_{\forall v, v \text{ is a consumer of } i} (s_v - s_u + 1) \Big) \cdot size(i) \tag{4}$$

where $size(i)$ is the size of argument i, which is defined in the given program by the type and the number of elements in the array corresponding to argument i. For example, in Figure 1 (a), the size of argument a_0 of $func\_a$ is 16*4=64 bytes when an integer takes up 4 bytes. We call node v a consumer of argument i of node u, if u's output corresponds to i is fed to v. An output argument can have more than one consumer like b_cd of $func\_b$ in Figure 1 (a) and (b). Similarly, the total amount of buffers needed due to all input arguments for a task v is

$$in\_bufs(v) = \sum_{\forall i \in \{\text{input args of } v\}} \Big(\max_{\forall d, d \text{ is a producer DMA of } i} (s_v - s_d + 1) \Big) \cdot size(i) \tag{5}$$

For an input argument of node v, we only consider producers that are DMAs. Otherwise, the buffers for output arguments of v's producer are also used for input arguments of v, since

```
if (···)
  memcpy(d_e_in[(i-s)%NUM_D_E_IN], d_e_0, SIZE);
else
  memcpy (d_e_in[(i-s)%NUM_D_E_IN], d_e_1, SIZE);

func_e(d_e_in[(i-s)%NUM_D_E_IN]);
```

Figure 3. A code snippet for merging two inputs from two PEs.

node v and v's producer are in the same PE. An input argument can have more than two producers due to the control flow of the program. For example, d_e of $func\_e$ has two producers $func\_d_0$ and $func\_d_1$ in Figure 1 (b). A DMA, a memory copy operation, is added after $func\_d_0$ and $func\_d_1$, before $func\_e$ to merge the the data from $func\_d_0$ and $func\_d_1$, as described in the code snippet in Figure 3.

In equation (5), a set of multiple buffers are introduced for an input argument. For example, for d_e of $func\_e$, $NUM\_D\_E\_IN$ buffers are generated in Figure 3. Let $b(i)$ denote the amount of data buffer that task i uses. Then, the sum of $out\_bufs(i)$ in (4) and $in\_bufs(i)$ in (5) is $b(i)$. Now, let $mem(j)$ denote the amount of local memory PE j has for data. The following inequality (6) gives constraints on the assignment of tasks to a PE based on the available data memory size on the PE.

$$\sum_{i=1}^{M} b(i) \cdot a_{ij} \le mem(j) \qquad \text{for all } j = 1, \cdots, P \tag{6}$$

As mentioned before, 0-1 variable a_{ij} indicates if node i is assigned to PE j or not. If we also treat $b(i)$ as a variable, equation (6) ends up with multiplications of variables. We attempt to keep $b(i)$ within a constant since we aim to define a linear program which can be solved within a time limit. To make $b(i)$ constant, first, $s_k - s_l$, $k,l = 1, \cdots, P$, in (4) and (5) need to be bound by constants. Second, DMAs should be decided for (5). Once we make $b(i)$ constant, we solve a ILP problem as follows.

$$\text{MPA-ii:} \quad \text{minimize } II, \quad \text{subject to} \quad (1), (2) \text{ and } (6). \tag{7}$$

MPA-ii stands for *Memory-constrained Processor Assignment for minimizing Initiation Interval*. To define a tractable memory-constrained processor assignment, we partition the problem into successive phases as described below. We solve this set of phase-ordered steps optimally to obtain a high-quality solution efficiently.

Step 1 conservative assumption: make a conservative assumption that all edges in stream graph G are across different PEs.

Step 2 initial stage assignment: assign the earliest possible stages to the nodes in G based on the assumption in **step 1** (Algorithm 1).

Step 3 buffer usage estimation: calculate the buffer usage of a node, $b(\cdot)$ using the stages calculated in **step 2**.

Step 4 MPA-ii: solve the ILP problem (7) using the $b(\cdot)$ in **step 3**.

Step 5 stage assignment optimization: based on the processor assignment solution of MPA-ii, a_{ij}, $i = 1,\cdots,M$, and $j = 1,\cdots,P$ of the ILP problem, reassign the stages so that the buffer usages decrease (Algorithms 2 and 3 in Section III-C).

If a producer node i and a consumer node j are in different PEs, our scheduler does not give them the same software pipelined stage. Otherwise, the execution of nodes i and j are not parallelized, but serialized though they are in different PEs. Thus, s_j must be larger than s_i. To come up with stages s_i to be used for $out\_bufs(\cdot)$ in (4) and $in\_bufs(\cdot)$ in (5), we conservatively assume that all edges in a stream graph are across two different PEs. In other words, we assume that for every edge (u,v) of G, there is a DMA-node d between u and v. Nodes u and v are called $head(d)$ and $tail(d)$, respectively. We fix the stage of a DMA-node d to $s_v - m$, when $v = tail(d)$. We give m a constant value zero when a DMA-node has the same stage with its consumer node, and 1, otherwise. Now the stages of two nodes u and v need to be apart at least by $1+m$. In stream graph, G, stages of all nodes in G and DMA-nodes corresponding the edges in G are obtained as described in Algorithm 1. The stage of a DMA-node generated from Algorithm 1 is guaranteed to be larger than that of its head and not to be larger than that of its tail. Note that the differences between stages in Algorithm 1 are

Algorithm 1 Initial stage assignment to define $b(i)$ for MPA-ii (in **step 2**).

```
Input: Stream graph $G$, and the stage difference between a DMA-node and its consumer, $m$.
Output: Stage $s$ for each node in $G$ and for each DMA-node.
  $L$ := the list of nodes in $G$ in the topological order;
  for all node $n$ in $L$ do
    $s_n := \max_{\forall p, p \text{ is a parent of } n}(s_p) + (1+m)$ ;
  end for
  for all DMA-node $d$ corresponding to an edge in $G$ do
    $s_d := s_{tail(d)} - m$;
  end for
```

sufficiently large. ILP constraints (1) and (2) are independent of software pipelined stages. Software pipelined stages are increased to exploit the parallelism when a producer and a consumer nodes are in different PEs. Algorithm 1 fully considers those parallelisms by always separating a producer and a consumer in different stages. Increasing stages more only unnecessarily increases the buffer usage.

Using the stages obtained so far, the usage of buffers $b(\cdot)$ is calculated as in (4) and (5) (**step 3**). Finally, the ILP problem (7) MPA-ii is solved (**step 4**). Depending on the actual processor assignment solution, a_{ij}, $i = 1,\cdots M$, and $j = 1,\cdots P$, there can be a consumer and a producer that are assigned to the same PE. If they are in the same PE, they do not need to be in different stages. Larger stage differences incur larger buffer usages. We reassign the stages of nodes based on the processor assignment solution a_{ij} so that we further decrease the usage of buffers (**step 5**). Section III-C describes this process. The stage assignment in **step 5** is an optimization that can be omitted. The stages from Algorithm 1 can be used without **step 5** because the processor assignment solution of MPA-ii does not change by **step 5**.

Note that the code size of a filter is constant. This constant value is currently ignored, but can be easily added to $b(\cdot)$ of inequality (6). If problem (7) is not feasible, we assume some portions of the data must be spilled to the global DRAM. Spilling data is one of our future work.

C. Stage Assignment Optimization

As the difference in stages between two nodes increases, the usage of data buffers increases. Further, the total number of pipelined stages determines the length of filling and draining phases of the whole pipeline, which is also an overhead of software pipelining. In **step 5**, we attempt to decrease these overheads by reducing the number of stages. First, we calculate the earliest possible stages based on the processor assignment solution of MPA-ii obtained in **step 4**. The detailed algorithm is given in Algorithm 2. A newly calculated stage of a node is always smaller than or equal to the initial stage obtained from Algorithm 1 because a consumer and a producer can have the same stage if they are assigned to the same PE.

However, these new stages cannot be used as they are. Let l_i and e_i denote the initial stage from Algorithm 1 and the new earliest possible stage from Algorithm 2, of node i, respectively. The value e_i is never larger than l_i. However, $(e_j - e_i)$ can be larger than $(l_j - l_i)$ for node i and j, which can lead to the violation of the conservative assumption on the amount of buffer usages $out\_bufs(\cdot)$ and $in\_bufs(\cdot)$ in equations (4) and (5). A stage l_i can decrease only to a stage s_i within the range of $[e_i, l_i]$ as long as the difference of the stages $s_j - s_i$ does not exceeds the initially assumed difference $l_j - l_i$, for any two nodes i and j in G or in DMA-nodes induced from MPA-ii. This is achieved by a backward traversal of the given stream graph G as described in Algorithm 3. Algorithm 3 always keeps the usage of buffers within the conservative assumption, and attempts to further reduce it. Figure 4 illustrates the stage assignment of the stream graph in Figure 1 (b) through Algorithms 1, 2, and 3, when m is 1. Note that the difference of stages between node $d1$ and the DMA-node after it increases from four to six by Algorithm 2 as shown in Figure 4 (a) and (b). The difference six is decreased back to four by Algorithm 3. Note that DMA-nodes over the edges (a,b) and $(b,d1)$ in Figure 4 (a) are eliminated in Figure 4 (c), reducing the number of total stages and the usage of buffers.

Finally, the resulting stages from Algorithm 3 are used. When there is no memory constraints, PA-ii in Section III-A is solved. Stage assignment is done only by Algorithm 2 after the processor assignments are obtained from PA-ii.

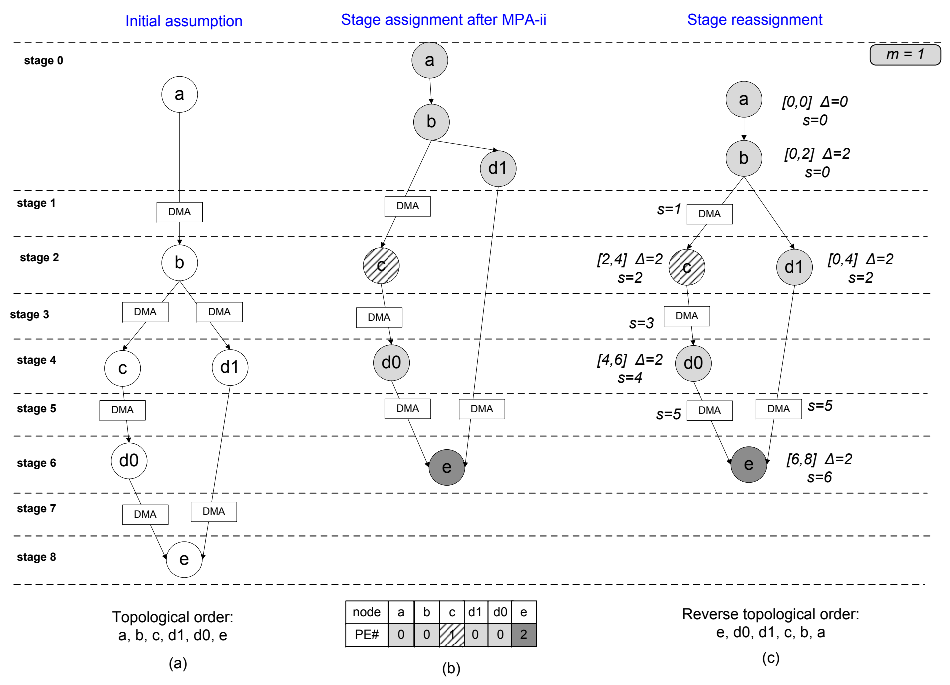


Figure 4. Stage assignment process when $m = 1$: (a) the initial stage assignment using Algorithm 1, (b) the stage assignment according to the processor assignment from MPA-ii using Algorithm 2, and (c) the final stage assignment using Algorithm 3. In (c), $[l_i, e_i]$, Δ_i and s_i are annotated to a node i as described in Algorithm 3.

Algorithm 2 Calculating the earliest possible stage based on the processor assignment of MPA-ii (in **Step 5**).

```
Input: Stream graph $G$, processor assignment, $proc(i)$ for all $i$ in $\{1, \cdots M\}$, and the stage difference between a DMA-node and its consumer, $m$.
Output: $s_n$ for all node $n$ in $G$ and for all DMA-node.
  $L$ := the list of nodes in $G$ in the topological order;
  $current\_stage$ := 0;
  for all node $n$ in $L$ do
    for all node $p$ which is a parent of $n$ do
      if $proc(p) \neq proc(n)$ then
        $st$ := $s_p + (1+m)$;
      else
        $st$ := $s_p$;
      end if
      if $st > current\_stage$ then
        $current\_stage := st$;
      end if
    end for
    $s_n := current\_stage$;
    for all node $p$ which is a parent of $n$ do
      if $proc(p) \neq proc(n)$ then
        Introduce a DMA operation between node $p$ and $n$ at $(s_n - m)$;
      end if
    end for
  end for
```

Algorithm 3 Post-fix on the stage adjustment (in **Step 5**).

```
Input: A range of possible stage $[e, l]$ for all nodes in $G$, where $e$ is the stage obtained from Algorithm 2, and $l$ is the initial stage from Algorithm 1. The stage difference between a DMA-node and its consumer, $m$.
Output: $s_n$ for all node $n$ in $G$ and for all DMA-node.
  $L'$ := list of nodes in $G$ in the reverse topological order;
  while $L'$ is not empty do
    $n$ := pop a node at the front of $L'$;
    $child\_delta$ := $\min(0, \min_{i \text{ is a child of } n}(\Delta_i))$;
    $\Delta_n$ := $\min(child\_delta, l_n - e_n)$;
    $s_n$ := $l_n - \Delta_n$;
  end while
  for all DMA-node $d'$ derived from the solution of MPA-ii do
    $s_{d'}$ := $s_{tail(d')} - m$;
  end for
```

D. Processor Assignment for Minimizing the Number of PEs under Real-time Constraints

Many streaming applications are real-time applications. Such applications impose one or more latency constraints between two tasks in applications.

In the software pipelined code, the actual latency from one task to another should not exceed the constraint given by the application. The actual latency from tasks i to j is a function of the stage difference between i and j and the initiation interval, II of the pipelined code. For example, in Figure 5, the latency from the beginning of $func\_a$ to $func\_c$ is $(2-0+1)*II$ when the stages of $func\_a$ and $func\_c$ are 0 and 2. The latency between two tasks whose software pipelined stages are s_1 and s_2, $(s_2 >= s_1)$ is, conservatively speaking, $(s_2 - s_1 + 1)*II$, where II is the initiation interval of the pipeline. Let $LAT = \{lat(i,j) | i,j = 1,\cdots,M\}$ be the set of latency constraints given from the application. The time between the completion of i and the start of j, $start\_time(j) - completion\_time(i) + 1$, must not be longer than $lat(i,j)$. Thus, $start\_time(j) - completion\_time(i) + 1 \leq (s_j - s_i + 1) \cdot II$. We obtain inequalities (8).

$$(s_j - s_i + 1) \cdot II \leq lat(i,j) \text{ for all } lat(i,j) \in LAT, i,j = 1,\cdots,M \quad (8)$$

We estimate the pipelined stages as described in **steps 1** and **2** in Section III-A to make $(s_j - s_i + 1)$ becomes constant. Then inequality (8) turns into a constraint on II such that,

$$II \leq \alpha, \quad \text{where } \alpha = \min_{\forall lat(i,j) \in LAT} \left(\frac{lat(i,j)}{s_j - s_i + 1}\right) \text{ is a constant.}$$

Consider the following problem.

$$\text{Minimize } II, \qquad \text{subject to } (1),(2),\text{and } (9) \qquad (9)$$

Problem (9) never produces smaller II than the II from PA-ii in Section III-A, because it has another constraint (9) added to PA-ii. What we can do best is comparing α in (9) to II from PA-ii to ensure if all the latency constraints can be met given PEs. Now we attempts to minimize the number of PEs to meet the given latency-constraints, which is represented by (9). It is a classic bin-packing problem, where a bin is a PE with a capacity α, and a task is an item with a weight $w(\cdot)$: given a bin size α and a list $w(1),\cdots,w(M)$ of sizes of items to pack, find a minimal integer P and P-partition $S_1 \cup \cdots \cup S_P$ of $\{1,\cdots,M\}$ such that $\sum_{i \in S_k} w(i) \leq \alpha$, for all $k = 1,\cdots,P$. This bin-packing problem is formulated as an ILP problem (10), LPA-numPE, standing for *Latency-constrained Processor Assignment for minimizing the Number of PEs*. A 0-1 integer variable a_{ij} is true if and only if task i is assigned to a PE j, where $i,j = 1,\cdots,M$. Another 0-1 integer variable y_j is true if and only if PE j is used.

$$\text{LPA-numPE}: \quad \text{minimize} \sum_{j=1}^{M} y_j \qquad \text{subject to} \qquad (10)$$

$$\sum_{j=1}^{M} a_{ij} = 1 \qquad \text{for all } i = 1,\cdots,M$$

$$\sum_{i=1}^{M} w(i) \cdot a_{ij} \leq \alpha \cdot y_j \qquad \text{for all } j = 1,\cdots,M$$

$$0 \leq a_{ij} \leq y_j \leq 1 \qquad \text{for all } i,j = 1,\cdots,M$$

This formulation gives a set of useful bounds: (1) a minimum number of PEs, LB_{PE}, that satisfies given timing constraints, and (2) a minimum number of PEs, UB_{PE}, for achieving the best possible II of the application, which is the largest workload among all kernels. Let the best possible II of the application be denoted by II_{best}. LB_{PE} is obtained solving LPA-numPE given the latency constraints, while UB_{PE} is obtained solving LPA-numPE by substituting α with II_{best}. Given a latency constraint in the form of inequality (9), it can always be met using at most LB_{PE} PEs for that constraint, if it is not smaller than II_{best}. If the constraint, more precisely, α of inequality (9), is smaller than II_{best}, there is no feasible system for satisfying the constraint. On the other hand, when the system has UB_{PE} PEs for an application, adding more PEs would not help improve the performance or meet any unsatisfied timing constraints.

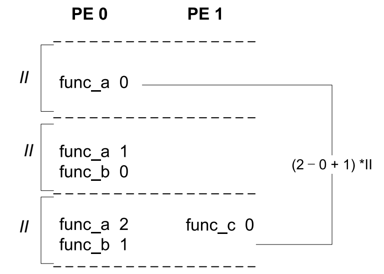


Figure 5. The relationship between the software pipelined stages and the latencies in time.

The bounds LB_{PE} and UB_{PE} can be used in solution space exploration for finding a schedule satisfying both timing and memory constraints. One scenario is shown in Figure 6. First, the range of PEs, $[LB_{PE}, UB_{PE}]$ that satisfies timing constraints is obtained by LPA-numPE. Under the assumption that the size of local memory per PE is fixed, the designer can increase the number of PE from LB_{PE} to UB_{PE} until a solution that can satisfy the memory constraints as well. If such a solution is found before P exceeds UB_{PE}, P is the smallest number of PEs that can meet the timing and memory constraints of the application. Once P becomes larger than UB_{PE}, it helps meet the memory constraint by providing more local memories, but does not improve the performance. Instead of adding more PEs, the designer can alternatively choose to increase the size of local memory per PE. Figure 6 illustrates one exploration example that does a sequential search on P, among many possible options. The designer can always take advantage of a binary search.

IV. Experimental Results

A. Experimentation Infrastructure and Benchmarks

The SPIR compiler is implemented within the SUIF compiler framework [13]. With our own modifications, we use the SUIF compiler to parse our input C language with keywords, to apply dependency analysis on array names and to generate multi-threaded code.

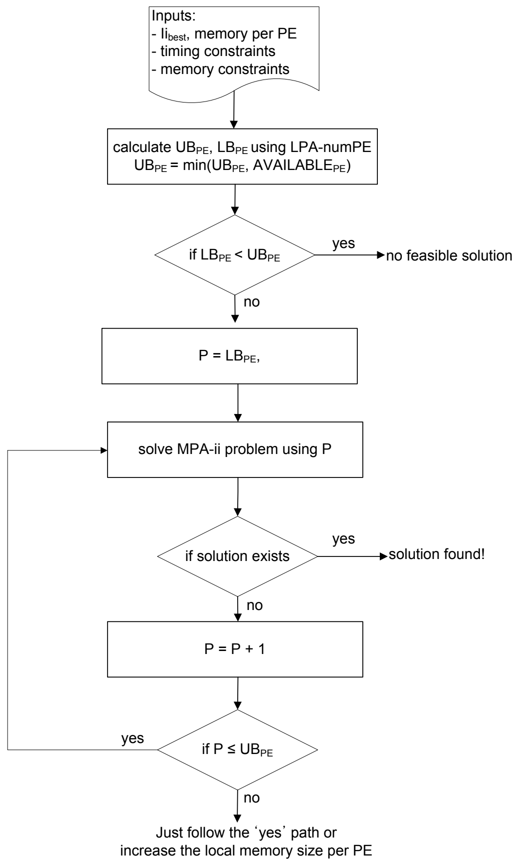

Inputs:
- II_{best}, memory per PE
- timing constraints
- memory constraints
calculate UB_{PE}, LB_{PE} using LPA-numPE
UB_{PE} = min(UB_{PE}, $AVAILABLE_{PE}$)
if LB_{PE} < UB_{PE}
yes
no feasible solution
no
P = LB_{PE},
solve MPA-ii problem using P
if solution exists
yes
solution found!
no
P = P + 1
yes
if P ≤ UB_{PE}
no
Just follow the 'yes' path or
increase the local memory size per PE

Figure 6. Exploring the design space for obtaining a solution satisfying both timing and memory constraints.

We evaluate our compiler with the *4G*, *WCDMA* and *DVB* protocols. *4G* is a next generation wireless protocol aimed at very high data communication rates. *WCDMA* (Wideband Code Division Multiple Access) is a common 3G protocol. *DVB* (Digital Video Broadcast) is a widely used protocol for digital media broadcasting. Each example consists of between 10 to 20 kernels of varying workload granularity of 100s to 1000s of cycles. We have assumed that these programs operate at a fixed data rate and they are expressed as a single stream graph.

We ran our experiments on a PlayStation3 equipped with a Cell processor, allowing us to measure runtimes as the number of PEs varied up to 6 processors.

B. Scalability of Our Memory-constrained Scheduler

We evaluate how well the performance of MPA-ii scales with the number of PEs. Figures 7 and 8 show the relative speed up over a single PE based on *II* and the actual execution times, respectively. Results are from our memory-constrained scheduler, MPA-ii. Local memory size per PE is given as 256 KB. Both graphs show the gradual increase of performance as the number of PEs increase and begin to lose the scaling factor after certain number of PEs. In Figure 7, the performance of *DVB* never increases after four PEs since it meets its bound on *II*, II_{best}, at four PEs. Likewise, *WCDMA* meets its bound on five PEs. Only the performance of *4G* increases up to twelve PEs. As mentioned in Section III-D these bounds on *II*, II_{best} can be easily found by solving problem LPA-numPE in (10).

The actual execution times illustrate the same trend. One exception is *WCDMA*, which does not scale well at three and five PEs. We note that *WCDMA* is the hardest to parallelize due to the granularity of its kernels. It has a few very large kernels with several small kernels. The workload of small kernels are even shorter than the workload of DMA operations, which hinders the hiding of the DMA operation latencies.

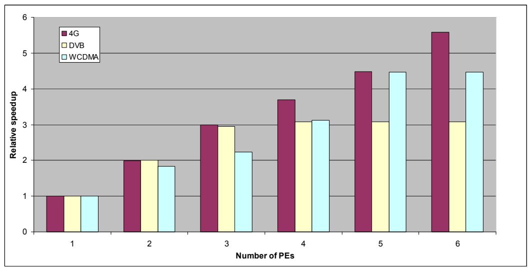


Figure 7. MPA-ii's relative predicted speedup in *II*, normalized to a single PE.

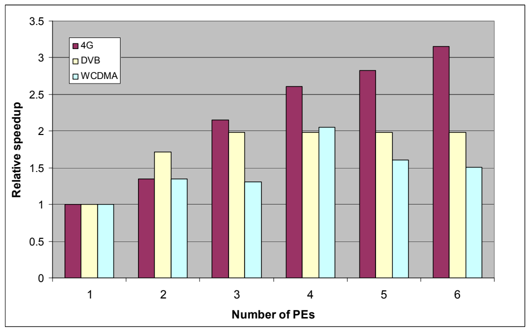


Figure 8. MPA-ii's relative speedup in execution time, normalized to a single PE.

C. Comparison Between the Memory-constrained Scheduler and Baseline Scheduler

This section evaluates the performance of our memory-constrained scheduler, MPA-ii, in finding solutions by comparing it against our baseline scheduler, PA-ii. PA-ii attempts to minimize *II* assuming unlimited local memory at each PE. For all three benchmarks, we observe that MPA-ii can find solutions that PA-ii could not find. Where both schedulers find a solution, the performance from both of them (based on II) is the same. That is, MPA-ii is able to find more solutions, without degrading the quality of the schedule. Figure 9 shows the comparison between MPA-ii and PA-ii

4G	number of PEs					
LMsize	1	2	3	4	5	6
32 KB						
64 KB						+
128 KB				+	+	+
256 KB	*	*	*	*	*	*
512 KB	*	*	*	*	*	*
1 MB	*	*	*	*	*	*

DVB	number of PEs					
LMsize	1	2	3	4	5	6
32 KB						
64 KB						
128 KB		+	+	+	+	+
256 KB	*	*	*	*	*	*
512 KB	*	*	*	*	*	*
1 MB	*	*	*	*	*	*

WCDMA	number processors					
LMsize	1	2	3	4	5	6
32 KB						
64 KB			+	+	+	+
128 KB		+	+	+	+	+
256 KB	*	*	*	*	*	*
512 KB	*	*	*	*	*	*
1 MB	*	*	*	*	*	*

Figure 9. The result of memory constrained scheduling: LMsize denotes the size of local memory per PE.

by marking each design point. A design point is composed of a number of PEs and the size of local memory per PE. A blank(' ') means there is no feasible solution so that none of the two schedulers can find a solution. An asterisk('*') denotes that both schedulers find feasible solutions. Finally, a plus('+') denotes that only MPA-ii finds a feasible solution while the solution by PA-ii is not able to meet the memory constraints.

D. Design Space Exploration with Latency Constraints

Section III-D shows that LB_{PE}, the minimum number of PEs for meeting latency constraints, is given by our latency-constrained scheduler LPA-numPE. Figures 10, 11, and 12 illustrate LB_{PE} for a certain range of II, for three benchmarks, respectively. They also show that II_{best} for each of them, denoted by blue lines, are met by 15, 4, and 5 PEs. In other words, 15, 4, and 5 are UB_{PE}s of the applications (green bars).

We can use this information as a starting point for exploring the design space comprised of the number of PEs, constraints on memory and the performance. Figure 13 shows the relative scheduling performance of the *WCDMA* protocol on a system supporting up to 5 PEs. The local memory size per PE is varied from 32 KB to 256 KB. The numbers represent the relative II. Smaller II means better performance. Suppose the latency-constraint is 30 in terms of II. The range $[LB_{PE}, UB_{PE}]$ is [2, 5]. One solution is 3 PEs and 64 KB per PE if the memory size is fixed as 64 KB. But, if the size of memory per PE is enlarged to 128 KB, 2 PE and 128 KB is another solution. It is shown that we can meet equivalent performance or still meet timing constraints, with a variety of system set-ups. The user can choose one as needed.

V. Related Work

Dataflow Scheduling. There has been numerous compilation projects on mapping dataflow graphs onto multi-core processors. Depending on the underlying dataflow model, some requires run-time system support, while others can generate compile-time schedules. In the MIT StreamIt compiler [14], the underlying model is based on the Synchronous Dataflow (SDF) model. They have examined the different static dataflow scheduling algorithms in [15], and their

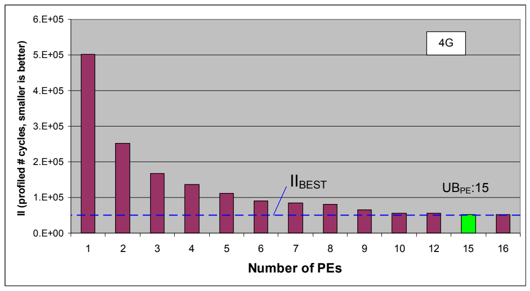


Figure 10. LB_{PE} and II_{best} of *4G*.

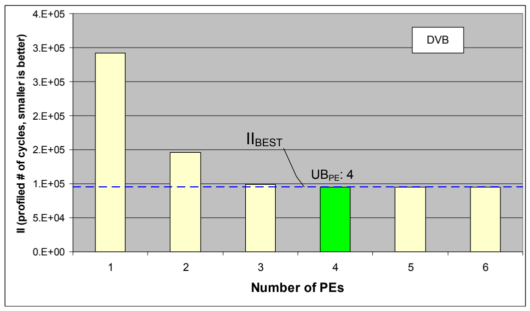


Figure 11. LB_{PE} and II_{best} of *DVB*.

impacts on the run-time execution. There are other projects, such as the Ptolemy [16], PeaCE [17], and DIF [18], that support multiple different dataflow models. This means that they cannot generate fully static run-time schedules, and have to provide run-time scheduler for run-time execution of dataflow actors.

Compilation Support for Multi-core DSP Processor. There has been numerous compilers for other multi-core DSP processors. The IBM Cell compiler is the most relevant to this study because its architecture [11] is the most similar to the Ardbeg processor architecture. Most of the IBM Cell compilation effort is focused on provided efficient single PE performance through various data-level parallelization techniques [19]. [20] advocates a multi-tier programming approach, which is similar to this thesis's proposed two-tiered compilation approach. However, it does not provide a compilation system that supports automatic code generations. More recent effort [10] has started to examine the function-

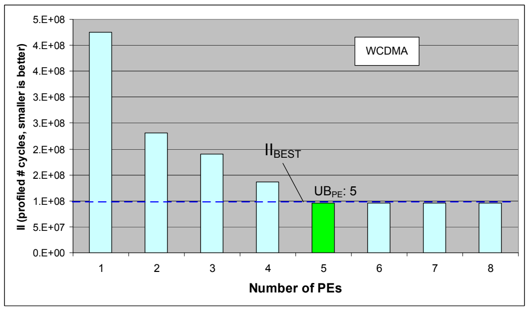


Figure 12. LB_{PE} and II_{best} of *WCDMA*.

WCDMA	number processors				
LMsize	1	2	3	4	5
32 KB					
64 KB			19	13	10
128 KB		22	19	13	10
256 KB	45	22	19	13	10

Figure 13. The relative *II* for different hardware set-ups: smaller is better.

level compilation methodology. There are other multi-core compilers that are not based on compiling dataflow models. These include the compiling the Brook streaming language onto multi-core processors [21], loop-centric parallelizing compiler for Vector-thread architecture [22], and basic-block level parallelization for the TRIP EDGE architecture [23].

Software Pipelining. In the compiler domain, modulo scheduling is a well known software pipelining technique [24]. There has been previous work purposing constraint-based modulo scheduling, including [25], and [26]. But all of these techniques are geared toward instruction-level modulo scheduling. [27] extends the modulo scheduling to software pipeline any loop nest in a multi-dimensional loop, which conceptually is similar to coarse-grained modulo scheduling. To our knowledge, there have not been any previous work exploring coarse-grained modulo scheduling for MPSoC architectures. However, the idea of coarse-grained software pipelineing has been explored before. [28] has proposed an algorithm that automatically breaks up nested loops, function calls, and control code into sets of coarse-grain filters based on a cost model. And, these sets of filters are then generated for parallel execution. [29] has proposed of using function-level software pipelining to stream data on the Imagine Stream Architecture. [9] also explored the idea of coarse-grained software pipelining on a tiled architecture.

VI. Conclusion

Stream programming is a natural programming model for many applications including SDR. This paper has introduced techniques for enabling stream compilation to embedded systems where timing and memory constraints are first class design considerations. We have demonstrated this by evaluating 3 protocols with our SPIR compiler and shown that our task scheduler produces scalable code that can meet a range of timing and memory constraints.

Acknowledgment

This work was supported by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD) (KRF-2007-356-D00200). This work was also supported from ARM Ltd.

References

[1] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A language for streaming applications," in *Proceedings of the 2002 International Conference on Compiler Construction*, Apr. 2002, pp. 179–196.

[2] I. Buck *et al.*, "Brook for GPUs: Stream computing on graphics hardware," *ACM Transactions on Graphics*, vol. 23, no. 3, pp. 777–786, Aug. 2004.

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

[4] D. Zhang, Z. Li, H. Song, and L. Liu, "A programming model for an embedded media processing architecture," in *Proceedings of the 5th International Symposium on Systems, Architectures, Modeling, and Simulation*, ser. Lecture Notes in Computer Science, vol. 3553, Jul. 2005, pp. 251–261.

[5] W. Mark, R. Glanville, K. Akeley, and J. Kilgard, "Cg: A system for programming graphics hardware in a C-like language," in *Proceedings of the 30th International Conference on Computer Graphics and Interactive Techniques*, Jul. 2003, pp. 893–907.

[6] 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 *Proceedings of the ACM SIGPLAN Conference on Programming Design and Implementation*, Jun. 2005, pp. 224–236.

[7] C. Consel *et al.*, "Spidle: A DSL approach to specifying streaming applications," in *Proceedings of the 2nd Intl. Conference on Generative Programming and Component Engineering*, 2003, pp. 1–17.

[8] E. Lee and D. Messerschmidt, "Synchronous Data Flow," in *Proceedings of the IEEE*, Sep. 1987, pp. 1235–1245.

[9] M. Gordon, W. Thies, and S. Amarasinghe, "Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs," in *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, Oct. 2006, pp. 151–162.

[10] M. Kudlur and S. Mahlke, "Orchestrating the Execution of Stream Programs on Multicore Platforms," in *2008 Conference on Programming Language Design and Implementation (PLDI)*, 2008, pp. 114–124.

[11] P. H. Hofstee, "All About the Cell Processor," in *Proceedings of IEEE Symposium on Low-Power and High-Speed Chips(COOL Chips VIII)*, April 2005.

[12] M. Woh *et al.*, "From SODA to Scotch: The Evolution of a Wireless Baseband Processor," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Nov. 2008, pp. 152–163.

[13] M. Hall *et al.*, "Maximizing Multiprocessor Performance with the SUIF Compiler," *IEEE Computer*, pp. 84–89, Dec. 1996.

[14] M. Gordon *et al.*, "A Stream Compiler for Communication-Exposed Architecture," in *Proceedings of Architectural Support for Programming Languages and Operating Systems*, 2002, pp. 291–301.

[15] M. Karczmarek, W. Thies, and S. Amarasinghe, "Phased Scheduling of Stream Programs," in *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems*, Jun. 2003, pp. 103–112.

[16] J. L. Pino, S. Bhattacharyya, and E. Lee, “A Hierarchical Multiprocessor Scheduling System for DSP Applications,” in *Proceedings of 29th Annual Asilomar Conference on Signals, Systems, and Computers*, 1995, p. 122.

[17] S. Ha *et al.*, “PeaCE: A Hardware-Software Codesign Environment for Multimedia Embedded Systems,” *ACM Transactions on Design Automation of Electronic Systems*, vol. 12 no. 3, Aug. 2007.

[18] W. Plishker, N. Sane, M. Kiemb, and S. S. Bhattacharyya, “Heterogeneous design in functional DIF,” in *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, Jul. 2008, pp. 157–166.

[19] A. E. Eichenberger *et al.*, “Using Advanced Compiler Technology to Exploit the Performance of the Cell Broadband Engine Architecture,” *IBM System Journal*, vol. 45, no. 1, pp. 59–84, 2006.

[20] R. Sakai *et al.*, “Programming and Performance Evaluation of the Cell Processor,” in *Hotchip 17*, Aug. 2005.

[21] S. Liao, Z. Du, G. Wu, and G. Lueh, “Data and Computation Transformations for Brook Streaming Applications on Multiprocessors,” in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2006, pp. 196–207.

[22] M. Hampton and K. Asanovic, “Compiling for Vector-Thread Architectures,” in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2008, pp. 205–215.

[23] A. Smith *et al.*, “Compiling for EDGE Architectures,” in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2006, pp. 185–195.

[24] B. R. Rau, “Iterative Modulo Scheduling: An Algorithm for Software Pipelined Loops,” in *Proceedings of 27th Annual International Symposium on Microarchitecture*, Nov. 1994, pp. 63–74.

[25] A. Eichenberger and E. Davidson, “Efficient Formulation For Optimal Modulo Schedulers,” in *Proceedings of Programming Language Design and Implementation*, June 1997, pp. 194–205.

[26] E. Altman and G. Gao, “Optimal Modulo Scheduling Through Enumeration,” *International Journal of Parallel Programming*, pp. 313–344, 1998.

[27] H. Rong *et al.*, “Single-Dimension Software Pipelining for Multi-Dimensional Loops,” in *Proceedings of the International Symposium on Code Generation and Optimization*, Mar. 2004.

[28] W. Du, R. Ferreira, and G. Agrawal, “Compiler Support for Exploiting Coarse-Grained Pipelined Parallelism,” in *Proceedings of ACM/IEEE Supercomputing Conference (SC)*, Nov. 2003.

[29] A. Das, W. Dally, and P. Mattson, “Compiling for Stream Processing,” in *Proceedings of Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2006.