Hierarchical Coarse-grained Stream Compilation for Software Defined Radio

Yuan Lin, Manjunath Kudlur, Scott Mahlke, and Trevor Mudge
Advanced Computer Architecture Laboratory
University of Michigan at Ann Arbor
{linyz, kvman, mahlke, tnm}@umich.edu

ABSTRACT

Software Defined Radio (SDR) is an emerging embedded domain where the physical layer of wireless protocols is implemented in software rather than the traditional application specific hardware. The operation throughput requirements of current third-generation (3G) wireless protocols are an order of magnitude higher than the capabilities of modern digital signal processors. Due to this steep performance requirement, heterogeneous multiprocessor system-on-chip designs have been proposed to support SDR. These heterogeneous multiprocessors provide difficult challenges for programmers and compilers to efficiently map applications onto the hardware. In this paper, we utilize a hierarchical dataflow programming model, referred to as SPIR, that is designed for modeling SDR applications. We then present a coarse-grained dataflow compilation strategy that assigns a SDR protocol's DSP kernels onto multiple processors, allocates memory buffers, and determines an execution schedule that meets a prescribed throughput. Unlike traditional approaches, coarse-grained compilation exploits task-level parallelism by scheduling concurrent DSP kernels instead of instructions. Because of the streaming nature of SDR protocols, we adapted an existing instruction-level software pipelining technique, modulo scheduling, for coarse-grained compilation. Our compilation methodology is able to generate parallel code that achieves near linear speedup on a SDR multiprocessor system.

Categories and Subject Descriptors

D.3.4 [**Processors**]: [Compilers]

Keywords

MPSoC Compilation, Modulo Scheduling, Software Defined Radio, Dataflow Programming Model

1. INTRODUCTION

In recent years, we have seen the emergence of an increasing number of wireless protocols that are applicable to different types of networks. Traditionally, the physical layer of wireless protocols is implemented with fixed function ASICs. Software defined radio (SDR) promises to deliver a cost effective and flexible solution by implementing the wide variety of wireless protocols in software. Such solutions have many potential advantages: 1) Multiple protocols can be supported simultaneously on the same hardware, allowing users to automatically adapt to the available wireless networks; 2) Lower engineering and debugging efforts are required for software solutions over hardware solutions; 3) Higher chip volume because the same chip can be used for multiple protocols, which lowers the cost; and 4) Better support for future protocol changes.

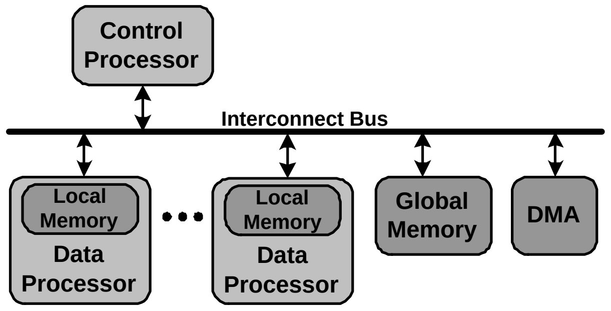


Figure 1: SDR control-data decoupled MPSoC architecture consisting of one general-purpose control processor, multiple data processors, and a hierarchical scratchpad memory system that are all interconnected with a bus.

Due to the high-throughput and low-power requirements, previous works have proposed using multiprocessor system-on-chip (MPSoC) digital signal processors (DSPs) to support SDR [8] [17]. These systems, as shown in Figure 1, fall under the category of control-data decoupled architectures. In control-data decoupled systems, functionality is separated into two classes of processors. Control processors are typically general-purpose processors that are capable of handling control-intensive code and are best suited for protocol scheduling and memory management. Conversely, data processors are specialized DSP processors that can perform heavy-duty data-intensive computations. Single-instruction multiple-data (SIMD) or vector processing is typically employed in the data processors. The system has a non-uniform memory architecture, with both a shared global memory and local memories on the data processors. Many systems use scratchpad memories instead of caches for local memories, which makes memory management the responsibility of the software. In many systems, one control processor is capable of supporting multiple data processors as shown in Figure 1.

Wireless protocols are collections of disparate DSP algorithm kernels that work together as one system. However, traditional DSP programming languages, such as C, are designed for stand-alone algorithms running on uniprocessor architectures. They assume a sequential programming model that is unfit for describing a wireless protocol's complex concurrent system-level behavior. Given that many embedded programs are still manually compiled by the programmers, compiling for a MPSoC SDR architecture is going to provide even greater challenges. One of the key advantages of SDR is the lower engineering effort for developing software over hardware, therefore a viable SDR solution must also provide programming language and compilation support that eases the software development effort.

SDR Programming Model. Previous work has proposed using the dataflow language to model streaming applications, including SDR protocols [21]. We implemented a third generation wireless protocol, Wideband Code Division Multiple Access (W-CDMA), as our SDR case study. We found that the protocol should be modeled with multiple decoupled dataflow streams expressed in a hierarchy, not as the single flat dataflow stream that was used in previous studies. This is because SDR protocols' inter-kernel communications have highly diverse streaming rates and patterns. In addition, traditional dataflow models express streaming patterns as FIFO queues of scalar variables. In SDR applications, many inter-kernel communications are queues of large meta variables, such as vectors and matrices. This requires a hierarchical communication model to express both the streaming patterns of the queues and the streaming patterns of the meta elements within the queues. In this paper, we present a hierarchical dataflow model, which effectively captures these streaming attributes.

Coarse-grained Compilation. Mapping a software implementation of wireless protocols onto MPSoC hardware requires software tool support. This SDR tool chain can be divided into two parts: the compilation of individual algorithm kernels for a DSP data processor, and the execution scheduling of the kernels for a MPSoC SDR system. This paper focuses on the second part of the tool chain — the MPSoC scheduling. For this study, the kernel compilation process is treated as a black-box, where the MPSoC scheduler assumes that a data processor compiler is provided by the MPSoC designers.

Previous work [16] has shown that the multi-processor scheduling problem can be divided into three major tasks: 1) processor assignment and memory allocation; 2) kernel execution ordering; and 3) kernel execution timing. All three tasks can be handled either statically by the compiler or dynamically by the run-time scheduler. In SDR protocols, the execution behavior is relatively static with limited run-time execution variations. The scheduling process needs to consider the inter-kernel communications, meet the real-time deadlines, and manage the scratchpad memories. This combination of factors favors a compile-time solution. Thus, we focus on designing a compiler for all three tasks. Coarse-grained function-level scheduling under strict memory constraints presents new challenges that have not been fully explored in previous compilation studies.

Coarse-grained Software Pipelining. Coarse-grained compilation requires function-level parallelism to utilize a MPSoC's resources. SDR protocols do not have many kernels that execute concurrently. They are streaming applications with coarse-grained pipeline-level parallelism. Software pipelining was proposed as a method to exploit the instruction-level parallelism by overlapping consecutive loop iterations. Stream computation can be viewed as a loop that iterates through each streaming data input, where the computation for successive data inputs can also be overlapped. The coarse-grained scheduling process is similar to instruction-level software pipelining, except that kernels and bulk memory transfers are scheduled onto processors and DMA engines, instead of scheduling instructions onto ALUs and memory units. In this study, we apply a well-known software pipelining technique, modulo scheduling, on a macro-level to schedule the kernels and bulk memory transfers onto the MPSoC system.

The contributions of this paper are two fold:

- A hierarchical dataflow programming model, SPIR, is presented that is designed to model wireless protocol's system-level streaming behavior.
- A coarse-grained modulo scheduling methodology is presented and evaluated that statically assigns SDR kernels onto processors, allocates scratchpad memories for inter-kernel communication, and creates a software pipelined execution schedule for the kernels.

The remainder of this paper is organized as follows. Section 2 describes our SDR case study, the W-CDMA wireless protocol, and its software characteristics. Section 3 describes SPIR and its design rationale. Section 4 explains the hierarchical dataflow compilation method. Finally, the effectiveness of the hierarchical compilation strategy is evaluated in Section 5, and we compare to related work in Section 6.

2. SDR CASE STUDY: W-CDMA

Wireless protocols are collections of disparate DSP algorithm kernels that work together as one system. In this study, we chose W-CDMA, as shown in Figure 2, as our case study to understand the software execution requirements for SDR systems. The top figure in Figure 2 shows the overall protocol diagram. The bottom figure shows the stream rates between kernels in W-CDMA receiver. For the sake of simpler illustration, the rates in this paper are expressed as a multiple of stream tokens. The actual size of a stream token is different for different kernels. For example, in LPF-Rx, a stream token is a complex number of two 12-bit fixed-point values. In Turbo decoder, a stream token is one 8-bit integer value. However, the sending and the receiving stream tokens of every pair of communicating kernels have the same token size. The following is a summary of the key observations.

Hierarchical algorithm description – Wireless protocols are systems connected together with DSP kernels. On the kernel level, each kernel consists of vector computations. The system level fits the dataflow execution model, whereas kernel level requires only a sequential execution model with vector arithmetic support.

DSP kernel macro-pipelining – Wireless protocols usually consist of multiple DSP algorithm kernels connected together in feed-forward pipelines. Data is streamed through kernels sequentially, resulting in almost no temporal locality.

Vector computations – Most of the computationally intensive DSP algorithm kernels have abundant data-level parallelism. Searcher, LPF, and Turbo decoder all operate on very wide vectors.

Meta-variable streaming – In addition to scalar variables, vectors and matrices are often passed between kernels. Some

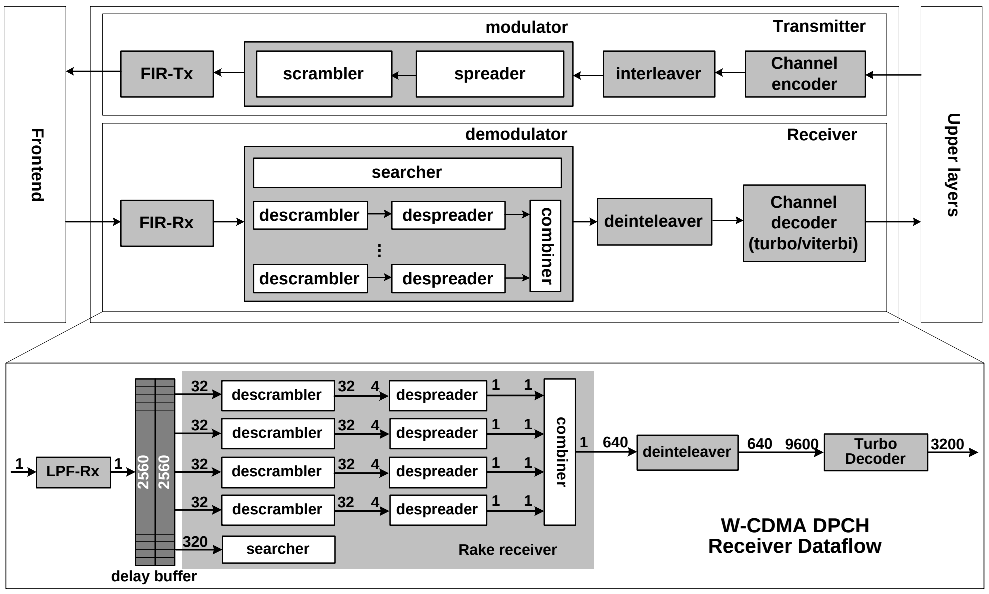


Figure 2: W-CDMA system diagram. The top diagram shows the W-CDMA protocol with the transmitter and receiver. The bottom diagram shows the W-CDMA DPCH (Dedicated Physical CHannel) receiver modeled as a dataflow. The source and destination nodes' stream rates are shown on the edges. Two 2560-wide vectors are required as the delay buffer between LPF and the Rake receiver. The dataflow stream rate also varies greatly between LPF's 1 to Turbo decoder's 9600. The rates are expressed as a multiple of stream tokens. The size of one stream token is different for different kernels. For example, a LPF-Rx stream token is a complex number of two 12-bit fixed-point values, and a Turbo decoder stream token is one 8-bit integer value.

of these meta-variables are relatively large in size and may be too large to fit onto a processor's local memory. This behavior requires a more complex streaming pattern than conventional scalar variable streaming. For example, as shown in Figure 2 dataflow, the input of the rake receiver requires two 2560-wide 32-bit vectors.

Large variations in streaming rate – While all kernels exhibit streaming characteristics, the streaming throughput rate maybe widely different between kernels in the same protocol. Some kernels, such as low-pass filters, can process input data individually. Other kernels, such as the interleaver, process data in large blocks with over 640 8-bit elements.

Real-time Deadlines – W-CDMA is a real-time application with periodic deadlines. As we will show in Section 4.2, these real-time deadlines can be translated into constraints on the modulo schedule.

3. SDR PROGRAMMING MODEL

The embedded market is already saturated with programming languages, such as Ada, C, Fortran, and Matlab, each with different language extensions and intrinsic libraries. All of these languages are based on the sequential programming model, which is a mismatch for SDR's MPSoC hardware and wireless protocols' streaming computations. However, because of the popularity of existing programming languages, it is unlikely that the DSP programmers are willing to adopt a completely new programming paradigm. Therefore, we designed SPIR (Signal Processing Intermediate Representation), a concurrent programming model, as an intermediate representation that can be automatically generated from existing high-level languages through a compiler frontend. However, the description of this frontend compilation process is beyond the scope of this paper.

SPIR represents a task graph consisting of a set of nodes (tasks) interconnected together with edges (dataflow). Each edge contains both input and output stream rates for the source and destination nodes. A node's stream rates correspond to the amount of data consumed and produced per invocation. Synchronous dataflow (SDF) is a restricted dataflow model where the stream rates are statically defined. This property allows a compiler to generate static execution schedules. SPIR supports a less restrictive dataflow model than the SDF, where dynamic stream rates are supported. However, in the context of this study, we assume the SDF model with static stream rates. Dynamic stream compilation strategies will be discussed in our future studies. SPIR supports hierarchy by allowing two types of nodes: a hierarchical node, which has a child graph made up of other SPIR nodes and a non-hierarchical node which does not have a child graph. Dataflow split and merge nodes are used to support data stream duplications and convergences. An example of the W-CDMA receiver's SPIR representation is shown in Figure 2's bottom diagram.

SPIR supports two forms of hierarchy, communication pattern and dataflow, which are described in the remainder of this section.

3.1 Hierarchical Communication Pattern

Vectors and matrices are often passed between DSP algorithms, resulting in complex streaming patterns that cannot be handled with a traditional one-dimensional FIFO stream buffer. On the top level, the communication pattern may be modeled as a buffer of vectors or matrices that are transferred between nodes. Within each FIFO element, vectors may have different element-wise access patterns. For example, a vector addition operation may be implemented by accessing each vector element sequentially. On the other

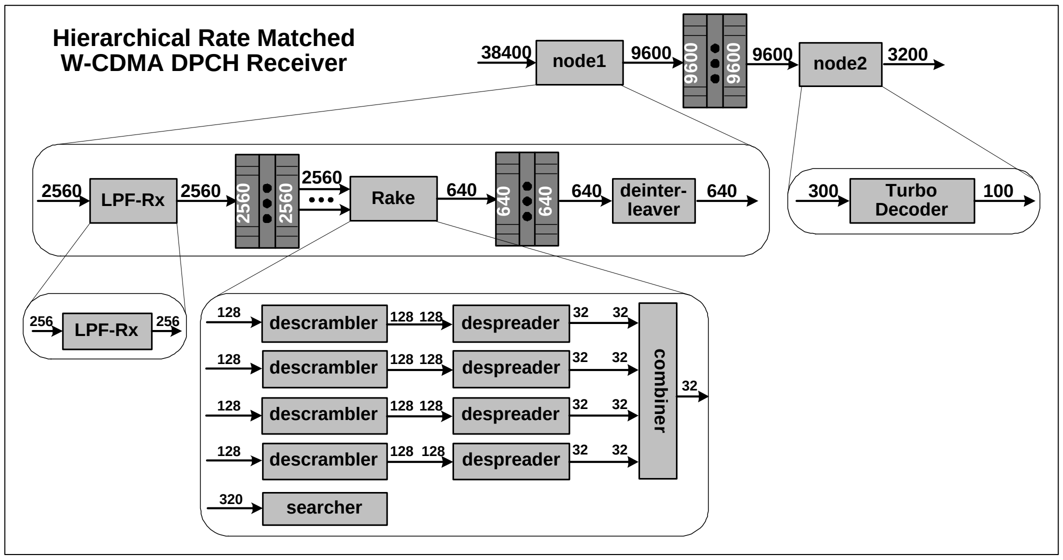


Figure 3: This graph shows W-CDMA DPCH receiver expressed as a rate matched hierarchical dataflow graph. Hierarchical dataflow allows kernels with widely different stream rates to be modeled as separate dataflow streams. It also allows delay buffers to be modeled hierarchically. Note that the hierarchy is defined by the user, but matched rates are the output of the SPIR compiler.

hand, a vector permutation will require access in the specific permutation pattern. In vector streaming, these are two independent streaming descriptions: 1) How are the vectors consumed and produced between nodes, and 2) How is each vector accessed by source and destination nodes? This motivates SPIR to model the software behavior with a hierarchical description, where each level of the hierarchy models a different type of streaming pattern.

One example that highlights the advantage of hierarchical communication is the DPCH receiver's dataflow between LPF-RX and Rake receiver shown in Figure 2. In W-CDMA, data are bundled in slots, with each slot containing 2560 data points. Due to multipath fading effect, the actual starting point for a slot can potentially be from any place within its previous slot. This requires the descrambler's input be placed in a delay buffer of at least 5120 elements, containing the current and previous slots. To make matters worse, there are multiple descramblers, each with a different starting point. Traditional dataflow models can only describe this as passing 2560 data points from LPF-Rx to Rake. On the other hand, a hierarchical vector buffer can also model the secondary communication pattern. On the top-level, it is modeled as a FIFO vector buffer between the FIR and Rake, where each element is a 2560-wide vector. On the bottom-level, each descrambler in the Rake receiver streams in data independently from its own starting point. Note that the memory is only deallocated at the top-level after each descrambler has streamed in 2560 data points, but data are accessed at the bottom-level.

In SPIR, vector and matrix buffers are described as special memory SPIR nodes. We currently support bounded FIFO and LIFO memory buffer descriptions. The edges that connect memory nodes in the same hierarchy layer describe push and pop operations to the memory buffer nodes. The child edges can only read or write to the memory buffers, but they are not allowed to push or pop buffer elements. This way, memory management of vector buffers is decoupled from the access patterns of each vector buffer element.

3.2 Hierarchical Dataflow

In SPIR's hierarchical stream graph representation, each dataflow node can contain its own dataflow graph. There are three major advantages of modeling hierarchical dataflow. The first reason is the hierarchical communication patterns mentioned in the previous subsection. The second reason is that SDR protocols' inter-kernel communications have widely different streaming rates. As shown in Figure 2, the input stream rates of DPCH receiver's LPF-Rx, deinterleaver, and Turbo decoder are 1, 640, and 9600, respectively. In this case, if we are to schedule the graph as one dataflow stream, then the stream rate of all three nodes must be matched to the highest stream rate of 9600. This results in unnecessarily large memory buffers for the LPF-RX and the deinterleaver. If the dataflow is modeled as a hierarchical graph, as shown in Figure 3, then the optimal stream rate of LPF-Rx, and deinterleaver can be determined independently from the Turbo decoder, allowing the compiler to find a more efficient schedule.

The third reason for the hierarchical dataflow is the need to model the dataflow sizes in addition to the dataflow rates. Many previous models assume a steady-state dataflow where the inputs to the stream receive an infinite number of tokens. However, this is not the case in SDR, where data are produced and processed in pre-defined data blocks. W-CDMA specifies each W-CDMA slot containing 2560 data points, 15 slots form one W-CDMA frame, and up to 5 frames are bundled into one W-CDMA TTI (Transmission Time Interval). Data across different TTIs cannot be streamed together, because consecutive TTIs may be transmitted using different W-CDMA channels. In addition, some algorithms must process data every slot, such as FIR and Rake receiver; some algorithms must process data on the frame boundary, such as the interleaver and deinterleaver; and other algorithms must process data on the TTI boundary, such as the Turbo

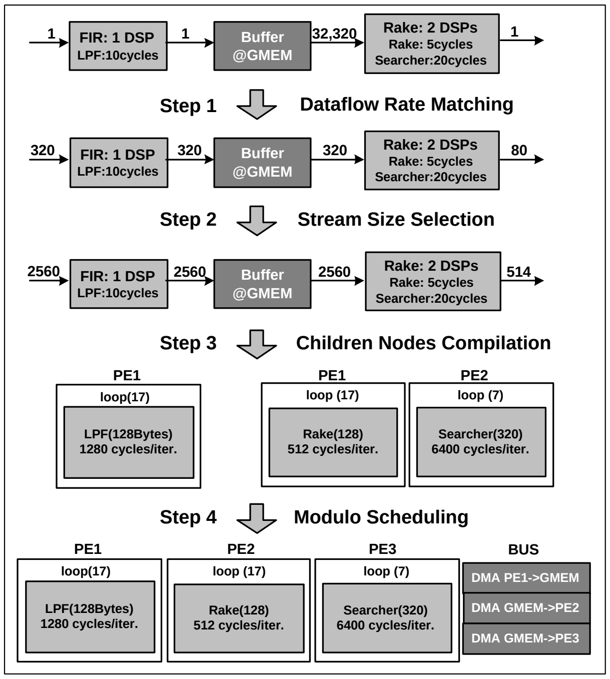


Figure 4: Hierarchical dataflow compilation summarized in 4 steps. In step 1, the dataflow is rate matched between each consumer/producer pair. In step 2, a multiple of the matched dataflow rate is chosen as the stream rate. In step 3, each hierarchical node is recursively scheduled. And in step 4, the graph nodes are software pipelined using coarse-grained modulo scheduling.

decoder. This leads naturally to a hierarchical graph representation, where each hierarchical node specifies the total amount of data streamed for its child graph.

4. COARSE-GRAINED COMPILATION

This section describes the hierarchical dataflow scheduling process. The input is the hierarchical SPIR dataflow graph as specified by the user in which rates on edges are potentially not matched. Hierarchical dataflow scheduling produces a software pipelined schedule in which rates are matched and data sizes for kernels are hierarchically chosen to fit the MPSoC's memory configuration. Note that, even though all the steps in the complete flow are outlined here, the main focus of this paper is the modulo scheduling algorithm used in software pipelining. All other steps are briefly described, the details of which are beyond the scope of this paper. Figure 4 shows the workflow of the scheduling method. Dataflow rate matching and stream size selection are done as preprocessing steps, prior to the actual hierarchical scheduling. The following subsections describe these steps. For the sake of simpler explanation, we first explain the basic coarse-grained modulo scheduling (step 4) before we explain issues that deals specifically with hierarchical scheduling (step 3).

4.1 Dataflow Graph Preprocessing

In this section, we first explain the DSP kernel compilation interface with the coarse-grained compilation. Then, the first two steps in Figure 4, are briefly explained.

Kernel Compilation and Profiling. Kernels form the building blocks for the SDR protocol dataflow graph. To make decisions for coarse-grained compilation, execution

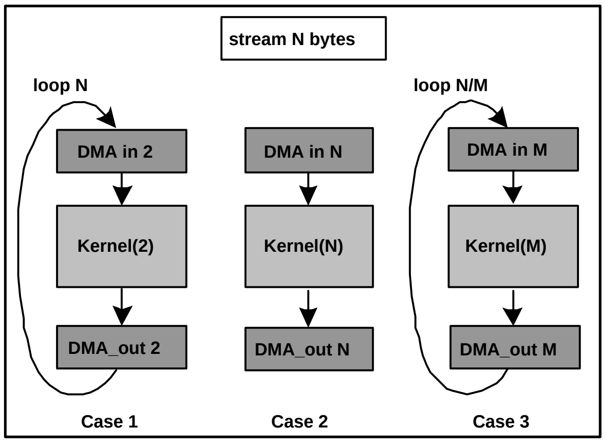


Figure 5: Stream size selection. In case 1, the stream size is too small, with very high overhead. In case 2, the stream size is too large, and the kernel cannot be software pipelined. The optimal stream size, M, is shown in case 3, striking a balance between the two extremes.

information about each kernel is required. Kernels are compiled and profiled individually on each of the processor types available on the MPSoC. Since each kernel can be instantiated with varying input sizes in the dataflow graph, profiling gathers information about the execution times of kernels with different input sizes. Kernel profiles are entered in a queryable format, so that later scheduling stages can easily access the information.

Rate matching and Stream size selection. The dataflow graph as specified by the programmer may have unmatched rates on the dataflow edges. To get a meaningful schedule, all rates have to be matched. The output of rate matching step is the *repetition vector*, with an entry for every kernel, and specifies how many times a kernel has to be repeated before the dependent kernels can begin execution. In this study, we assume the synchronous dataflow (SDF) model. There exists a large body of previous work for SDF rate matching algorithms [2]. We implemented one of these algorithms which finds the schedule with the minimum buffer sizes. This schedule is the least repetition vector. Note that the least repetition vector may not be the best option in terms of memory utilization and DMA traffic. The stream size selection step chooses a multiple of the repetition vector so that data transfer between kernels happens at a coarse granularity to amortize the DMA transfer overhead. Figure 5 illustrates the stream size selection problem. In case 1, the kernel is repeated twice. Therefore, for each invocation of the kernel, 2 bytes of data is transferred in and out of the processor, which results in high DMA startup overhead. In case 2, the kernel is repeated N times. Even though the DMA overhead is amortized over N bytes, the processor may not have enough memory to hold the data required for N invocations of the kernel. Case 3 strikes a balance by choosing a repetition factor of M. The DMA overhead is amortized for N/M transfers, and at the same time, only M bytes are transfered which could fit in the processor's memory. We currently perform a binary search through all possible values of M for a dataflow graph to determine the optimal value based on shortest execution latency.

4.2 Coarse-grained Modulo Scheduling

In traditional compilation, software pipelining is a technique to extract instruction-level parallelism by overlapping the execution of operations from successive loop iterations. In coarse-grained kernel scheduling, stream computation can be viewed as a loop that iterates through each streaming data input, where the computation for successive data inputs can also be overlapped. Modulo scheduling [19] is a well-known software pipelining algorithm that can achieve very good solutions. In this section, we present a coarse-grained modulo scheduling algorithm used to schedule a rate matched hierarchical dataflow graph on to a MPSoC. Similar to instruction-level modulo scheduling, coarse-grained modulo scheduling has to honor resource and dependency constraints between dataflow nodes. However, coarse-grained modulo scheduling differs from traditional modulo scheduling in the following ways.

Storage assignment. In traditional modulo scheduling, allocation of storage (e.g., rotating registers) used for carrying values between operations is performed as a post-processing step. Enough storage is assumed to be available during the scheduling phase, while the register allocation phase does the actual storage allocation. In coarse-grained modulo scheduling, memory buffers must be allocated on the processors where dataflow nodes are scheduled. Typically, the local memory available on processors is limited. Also, MPSoCs can have processing elements with varying memory capacities. This limited non-uniform distribution of memories makes the storage assignment a first-class scheduling constraint. Postponing storage assignment to a later phase results in the scheduler making aggressive decisions about node placements on processors. Consequently, storage assignment fails in many cases. Therefore, in the coarse-grained modulo scheduling method presented, scheduling and storage assignment are performed in a single phase.

Scheduling data movement. Traditional modulo scheduling assumes that the value written to the register by an operation is available to dependent operations in the very next cycle. This is because the register file is connected to all function units. However, in a MPSoC, processors have their own local memories and the data is transported between processors. DMA operations used for moving the data between processors take significant amount of time, and dependent operations must wait for the DMAs to complete before they can begin execution. Thus, unlike traditional modulo scheduling, the coarse-grained modulo scheduler must explicitly schedule the DMA operations used for moving data between processors.

II Selection. In modulo scheduling, II (initiation interval) is the interval between the start of successive iterations. The minimum initiation interval (MII) is defined as $MII = Max(ResMII, RecMII)$, where ResMII is the resource constrained MII, and RecMII is the recurrence constrained MII. In coarse-grained modulo scheduling, ResMII is defined by the total latency of all nodes in the graph divided by the number of processors allocated to the graph. RecMII is defined by the maximum latency of each feedback path.

Since SDR protocols are real-time applications, the scheduler must also take timing constraints into consideration. In W-CDMA, the timing constraint is defined by the overall data throughput, which is 2Mbps as the output rate of the receiving data channel. If we use the matched data rates shown in Figure 3, then at the top hierarchical level, the maximum II must be 610K clock cycles on a MPSoC with 400MHz data processors as an example. Like instruction-level modulo scheduling, the II selection process starts at MII, and is iteratively increased until all of the nodes are scheduled or the maximum II is reached. If the maximum II is reached and no valid schedule is found, then a failure message is returned.

For each II, the modulo scheduler assigns nodes to processors and allocates DMA buffers for producer/consumer node pairs that are assigned to different processors. In the following two subsections, we present two modulo schedulers. The first uses a greedy heuristic which naively assigns nodes to processors based only on execution latencies. We then present a SMT(Satisfiability Modulo Theory)-based modulo scheduler, which takes data processors' scratchpad memory size constraints into consideration.

4.2.1 Greedy Scheduling

In greedy scheduling, nodes are scheduled sequentially in each hierarchical level based on their priorities. The hierarchical graph is scheduled bottom-up, starting from the lowest hierarchy layers with only non-hierarchical nodes. Each node is scheduled onto the processors with the lightest workload. The priority is based on hierarchy, with the hierarchical nodes have higher priority over non-hierarchical nodes. Within hierarchical nodes, the priority is based on the number of child nodes, then the number of processors allocated to the node, and finally by the execution latency of the node. Non-hierarchical nodes are only allocated to one processor, so they are prioritized only by execution latency. After nodes are assigned to processors, DMA operations are assigned to each SPIR edge where the source and destination are on different processors. DMA operations are also assigned if the source or the destination node is a memory SPIR node.

4.2.2 SMT based Scheduling

This section describes an exhaustive modulo scheduler that forms resource constraints and uses a Satisfiability Modulo Theory (SMT) solver — Yices [5] to get a valid modulo schedule. The input is a rate matched hierarchical dataflow graph, and the output is assignment of dataflow nodes to processors, dataflow edges to DMA engines, and the allocation of buffers on processors' local memories. Hierarchical nodes are recursively scheduled before the graph which contains the hierarchical node is itself scheduled. Because the local schedules of a hierarchical node can use more than one processor, each hierarchical node is broken into multiple sub-nodes such that each sub-node's schedule only uses one processor. Issues related to hierarchical modulo scheduling are explained in the next section. The following descriptions assumes this sub-node decomposition is already done by our compiler, and each node is assigned to only one processor.

Consider a dataflow graph $G = (V, E)$, with P nodes and Q edges. Associated with every dataflow node $v_i \in V$ is $exec\_time(v_i)$, the time required to execute the node. Associated with every dataflow edge $e_i = (u, v) \in E$ is $data\_size(e_i)$, the amount of data in bytes produced by node u and consumed by node v per invocation. Let N be the number of data processors in the SDR MPSoC, N boolean variables $a_{ij}, j \in \{1, N\}$ are introduced for every node v_i to denote the status of node v_i assigned to the processor j. To ensure each node is assigned to one and only processor, the following boolean constraints are asserted.

$$\begin{aligned}&\bigvee_{j=1}^{N} a_{ij} == true && i \in \{1, P\}\\ &a_{ij_1} \wedge a_{ij_2} == false && 1 <= j_1 < j_2 <= N, j_1 \neq j_2\end{aligned} \tag{1}$$

Let II be the initiation interval, the number of cycles between successive instantiations of the dataflow graph instances. Since every processor will have to repeat its jobs every II cycles, the total time taken by the nodes assigned to a processor should not exceed II. The following constraint ensures that.

$$\sum_{i=1}^{P} a_{ij} \times exec\_time(v_i) <= II \qquad j \in \{1, N\} \tag{2}$$

Consider an edge $e_i = (v_p, v_c)$. If the producer node v_p and consumer node v_c are assigned to different processors, a DMA has to be scheduled to transfer the data. Let D be the number of simultaneous DMA operations supported by the DMA engine in the system. D boolean variables d_{ij} are introduced for every edge e_i to denote the fact that the data transfer corresponding to the edge uses up one of the available DMAs. The following assertions are added to ensure that either v_p and v_c get assigned to the same processor, or a DMA is reserved for the data transfer. $t_j, j \in 1, N$ expresses the status of the producer p and consumer c assigned to the same processor j, and $t_{N+j}, j \in 1, D$ expresses the status of a DMA operation assigned on DMA engine j.

$$\begin{aligned}t_j &= a_{pj} \wedge a_{cj} && j \in 1, N\\ t_{N+j} &= d_{ij} && j \in 1, D\\ &\bigvee_{j=1}^{N+D} t_j == true\\ t_{j_1} &\wedge t_{j_2} == false && 1 <= j_1 < j_2 <= N + D, j_1 \neq j_2\end{aligned} \tag{3}$$

Let $dma\_time(e_i)$ denote the time required by the DMA engine to transfer the data corresponding to edge e_i. $dma\_time(e_i)$ would depend on $data\_size(e_i)$, the DMA startup overhead, bus width and bus latency. Since the DMA engine has to repeat its job every II cycles, the amount of time occupied by all the edges assigned to a DMA should not exceed II. This is guaranteed by the following constraint.

$$\sum_{i=1}^{Q} d_{ij} \times dma\_time(e_i) <= II \qquad j \in \{1, D\} \tag{4}$$

Consider an edge $e_i = (v_p, v_c)$, with v_p and v_c assigned to different processors. When the producer v_p finishes execution, its output buffer has to be transfered to the processor on which v_c is assigned. Until the DMA transfer is complete, next instance of v_p cannot start writing to the same output buffer. For overlapping the DMA transfer and the next instance of v_p, two buffers have to be allocated for storing the output of v_p. Symmetrically, the DMA transfer to the v_c processor and the execution of v_c can be overlapped only if two buffers are allocated to store the input to v_c. For every edge e_i, integer variables bp_i and bc_i are introduced to denote the number of buffers allocated on the input side and output side of the DMA respectively. The following constraint ensures that enough buffers are present on the input and output sides to guarantee that the given II is achievable.

$$\begin{aligned}same\_proc &= \bigvee_{j=1}^{N} a_{pj} \wedge a_{cj}\\ dma_p &\Leftrightarrow (dma\_time(e_i) + exec\_time(v_p) <= II \times bp_i)\\ dma_c &\Leftrightarrow (dma\_time(e_i) + exec\_time(v_c) <= II \times bc_i)\\ \neg same\_proc &\rightarrow (dma_p \wedge dma_c) == true\end{aligned} \tag{5}$$

The processor on which a node v_i is assigned should have enough local memory to contain all the input and output buffers needed by v_i. Let $mem\_size(p)$ be the size of local memory available on processor p. If a node v_i is assigned to p, the following assertion ensures that the local memory available on p is greater than the size of buffers allocated for all incoming and outgoing data.

$$\begin{aligned}m1 &= \sum_{j \in in\_edges(v_i)} a_{ip} \times bc_j \times data\_size(e_j)\\ m2 &= \sum_{j \in out\_edges(v_i)} a_{ip} \times bp_j \times data\_size(e_j)\\ m1 + m2 &<= mem\_size(p)\end{aligned} \tag{6}$$

Note that the equation involves a product of a boolean variable and an integer variable. Yices allows modeling such an expression using *If-then-else* construct. For example, product of boolean variable `b` and integer variable `x` can be modeled as `(ite b x 0)`.

Solving for boolean variables a_{ij}, d_{ij} and integer variables bc_i and bp_i under the constraints given by equations 1 through 6 produces a legal modulo schedule with initiation interval II for the graph G. A SPIR dataflow graph may contain hierarchical nodes which themselves are dataflow graphs. Hierarchical nodes are recursively traversed and scheduled using the same formulation as above. Scheduling decisions are passed up to the higher levels of hierarchy where decisions from multiple hierarchical nodes are combined to form the modulo schedule for the entire graph.

4.3 Hierarchical Scheduling

In this section, we will briefly discuss a few issues that are specific to the hierarchical modulo scheduling.

Virtual Resource Assignment. The processor and DMA assignments for each child node cannot be tied to physical resources because scheduling the parent node may alter its child nodes' schedules. For example, if a child node has two kernels, A and B, that are assigned to two different processors, and a DMA is generated for the data transfer. The scheduler for the parent node may choose to put A and B on the same processor due to other resource constraints. And the DMA operations must be replaced with a memory move operation to ensure execution correctness. Physical resources are only assigned at the root level of SPIR.

Hierarchical Synchronization. In a hierarchical SPIR graph, each hierarchical node is modulo scheduled before the graph itself is scheduled. Therefore, each hierarchical node contains its own independent modulo schedule, which may occupy multiple hardware resources. Within a modulo schedule, the start and end times of each resource must be the same to prevent data race conditions. In a hierarchical schedule, execution of one modulo schedule may be blocked even if all of its resources are free, because one of its resources is waiting to synchronize as part of another modulo

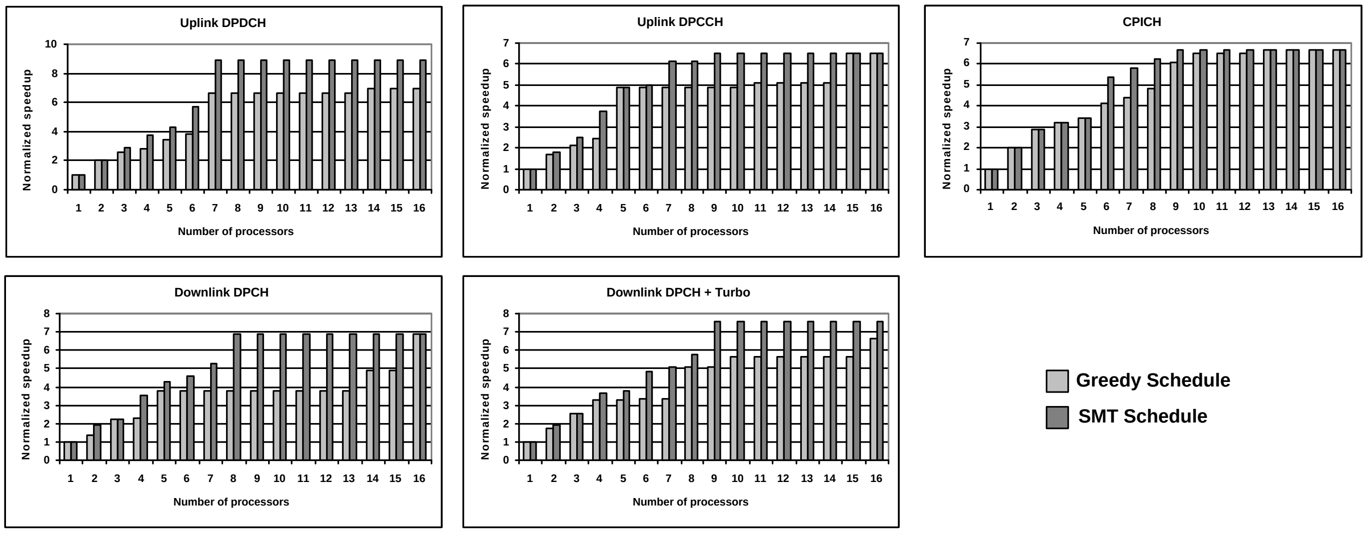


Figure 6: Execution speedup for W-CDMA benchmarks compiled by greedy and modulo schedulers running on 1 to 16 data processors.

schedule. Too many layers of hierarchy will create unnecessary synchronization overheads, which can be detrimental to the overall performance.

Prolog and Epilog Generation. In coarse-grained software pipelining, the prolog and epilog overheads are much greater due to the DMA transfers between the kernels. For a N stage pipeline, up to $N + 1$ stages of DMA transfers will be inserted, which requires a maximum of $2N$ stages for each of the prolog and epilog. For a hierarchical modulo schedule, executing its children nodes' prolog and epilog for each invocation requires too much overhead. Therefore, prolog and epilog generations are postponed until after the entire hierarchical graph is scheduled to reduce the overall execution overhead.

4.4 Post Scheduling Compilation

The SDR MPSoC system includes one control processor and multiple data processors. The DSP kernels are executed on the data processors. The multiprocessor schedule is executed on the control processor through a set of Remote Procedure Calls (RPCs) and Directed Memory Access (DMA) operations. Inter-processor synchronizations are also managed centrally by the control processor.

The final step of the compilation process converts the schedule into a set of finite states. This execution schedule consists a list of remote procedure calls (RPCs), DMAs, and a set of synchronization barriers. Each RPC and DMA is translated into a pair of states: the first state issues the RPC/DMA call on the target data processor, and the second state waits for its completion. A synchronization barrier is the state when all of the synchronizing resources reach their second states. The output of this process is a C file, which is then compiled onto the control processor by its native compiler.

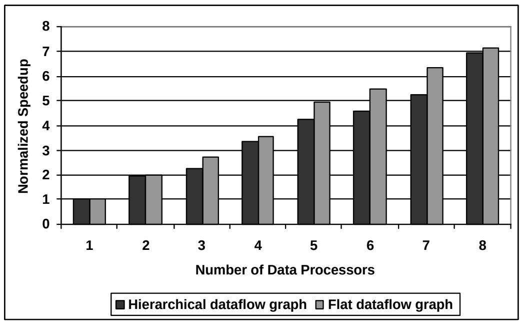
5. EXPERIMENTATION

The compilation framework is implemented on top of the SUIF compiler infrastructure [11]. In this study, SPIR benchmarks are written by hand. We are currently implementing a frontend parser in SUIF that translates C to SPIR. The SPIR code is compiled by our SPIR compiler to generate a coarse-grained modulo schedule. The schedule is then converted back into SUIF IR, and C code is generated through a SUIF-to-C backend. The final C code is then compiled and simulated to generate system-level execution profile. The simulator is designed for SODA [17], a MPSoC architecture for SDR. SODA has an ARM control processor, multiple SIMD data processors, and a 64KB global scratchpad memory. Each data processor has a 12KB local scratchpad memory. The processors and the global memory are connected together through a 200MHz 32-bit shared bus. Our modulo scheduler's SMT solver is implemented using the Yices C library [5].

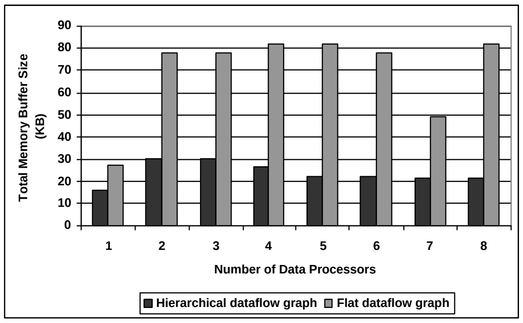
5.1 SDR Case Study: W-CDMA

W-CDMA protocol specifications [12] define multiple transmission modes for different purposes, ranging from data and voice transmissions to synchronization. In this study, we picked five operating modes that cover the essential W-CDMA operations, and handcoded them in SPIR. These five modes are: downlink DPCH(Dedicated Physical CHannel); uplink DPCCH(Dedicated Physical Control CHannel); uplink DPDCH (Dedicated Physical Data CHannel); and CPICH (Common Pilot CHannel). Downlink DPCH is the main data receiver channel, it is time-multiplexed between receiving protocol control and user data. We included two versions of the downlink DPCH, one with the Turbo decoder and one without. This is because many proposed SDR solutions still use Turbo ASIC accelerators [22] due to its high computation requirements. Uplink DPCCH and DPDCH are the transmitter counter-parts of the downlink DPCH. And finally, CPICH is the synchronization channel, which is used to measure signal strength and synchronize data transmission.

Greedy vs. SMT Modulo Scheduling. Figure 6 shows the overall execution speedup for the W-CDMA benchmarks compiled with the two schedulers, running on 1 to 16 data processors with 1 control processor. The execution speedup is normalized to the execution time of the benchmarks running on 1 data processor. For all of the bench-



(a) Downlink DPCH's normalized execution latency speedup on a hierarchical dataflow versus flat dataflow graph



(b) Downlink DPCH's total DMA memory buffer sizes on a hierarchical dataflow versus flat dataflow graph

Figure 7: Comparisons between compilation of a hierarchical dataflow graph versus a flat dataflow graph for downlink DPCH.

marks, the SMT scheduler achieves near-linear speedup up to 8 processors. However, it cannot efficiently utilize more than 10 processors, even though there are many more kernels in the benchmarks. The reason is because there are a few bottleneck algorithms, such as filter, searcher, and Turbo decoder, that require much more computational resources then the rest of the algorithms. Therefore, even though there are many processors available, the majority of the time are spend waiting for the bottleneck algorithms to finish. Compared to the SMT scheduler, the greedy scheduler is able to achieve equal speedup on a few benchmarks, such as the uplink DPCCH running on 5 and 6 data processors. On average, the greedy scheduler achieves 20% less speedup than the SMT scheduler.

Flat vs. Hierarchical Dataflow. In W-CDMA, data are divided into TTI (Transmission Time Interval) blocks. Inter-TTI data blocks cannot be pipelined because each TTI may be operating in a different transmission mode. Each TTI block contains a maximum of 5 W-CDMA frames. Based on the W-CDMA protocol specification, the interleaver and deinterleaver need to buffer data as frame blocks. If we rate-match the flat graph, then all of the kernel computations are processed with the stream rate of one W-CDMA frame block. Therefore, one cannot generate a software-pipelined schedule for most W-CDMA transmission modes with a flat graph. The only successful benchmark is the downlink DPCH, because it does not use the interleaver or deinterleaver. The comparison results for this benchmark are shown in Figure 7. In Figure 7(a), the execution latency for the software pipelined flat dataflow graph is lower than the hierarchical dataflow graph. This is due to the addition synchronization overhead for the hierarchical module schedules. However, as shown in Figure 7(b), the DMA memory buffers allocated for the flat dataflow graph is 2 to 3 times higher than the hierarchical dataflow graph. This is because the hierarchical dataflow can independently optimize the data buffer size at each level. The results show that hierarchy should be used sparingly, as excessive number of hierarchical layers can result in unnecessary performance degradation. However, hierarchical dataflow graphs can achieve more efficient memory utilization over flat dataflow graphs. Hierarchy also allows more dataflow graphs to be software pipelined.

6. RELATED WORK

Dataflow Languages. There have been many previous studies on dataflow programming model and languages. Dataflow was first proposed as the Kahn process network [13]. In Kahn's model, network nodes communicate concurrently through unidirectional infinite-capacity FIFO queues. Each network node contains its own internal state. Reading from the FIFO queues is blocking, and writing to the FIFO queues is non-blocking. Because of the blocking-read operation, the context switching overhead is high. Researchers have later proposed dataflow process networks [14], which are a special case of the Kahn network. In dataflow process networks, the communication rates (firing rules) between network nodes (actors) are explicitly defined. Many variations of dataflow processor networks have been proposed. One of the most popular is the Synchronous Dataflow (SDF), in which the firing rates are static. A great amount of work on dataflow process network has been done as a part of the Ptolemy project [15]. They have developed an extensive software framework for dataflow modeling and simulations. In terms of compilation support, MIT's StreamIt [9] language is modeled after the synchronous dataflow model, and a dataflow compiler was designed for tiled-based architectures.

Software Pipelining. In the compiler domain, modulo scheduling is a well known software pipelining technique [19]. There has been previous work purposing constraint-based modulo scheduling, including [7], and [1]. But all of these techniques are geared toward instruction-level modulo scheduling. [20] 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 pipelining has been explored before. [6] 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. [4] has proposed of using function-level software pipelining to stream data on the Imagine Stream Architecture. The problems of kernel profiling and stream size selection are also discussed in this paper. [10] also explored the idea of coarse-grained software pipelining on a tiled architecture.

Hierarchical Modeling and Compilation. Hierarchical dataflow models have been proposed before to model multi-rate DSP applications with constraints [3]. [23] has also noted that traditional 1-dimensional streams description are not efficient for modeling complex DSP streaming behavior. [18] has proposed a hierarchical multiprocessor scheduling algorithm. Because the algorithm didn't target toward a specific processor, it is not directly applicable as a SDR scheduling solution.

7. CONCLUSION

In this study, we proposed a hierarchical dataflow programming model for describing Software Defined Radio wireless protocols. We then proposed a coarse-grained compilation strategy for scheduling SDR applications onto MPSoC architectures. Because coarse-grained compilation requires function-level parallelism, we adapted a well known instruction-level software pipelining technique, modulo scheduling, to exploit SDR applications' coarse-grained pipeline-level parallelism. We used W-CDMA as our SDR case study, and developed a set of key W-CDMA operation modes in our programming model. Our results have shown that our compiler is able to generate multi-processor schedules that get near linear speedups for various MPSoC system configurations, while dealing effectively with the tight memory constraints of embedded MPSoC systems.

8. ACKNOWLEDGMENT

Yuan Lin is supported by a Motorola University Partnership in Research Grant. This research is also supported by ARM Ltd., the National Science Foundation grant NSF-ITR CCR-0325898 and CCR-0325761.

9. REFERENCES

[1] E. Altman and G. Gao. Optimal Modulo Scheduling Through Enumeration. In *International Journal of Parallel Programming*, pages 313–344, 1998.

[2] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Synthesis of Embedded Software from Synchronous Dataflow Specifications. In *Journal of VLSI Signal Processing Systems*, volume 21, no. 2, pages 151–166, June 1999.

[3] N. Chandrachoodan and S. S. Bhattacharyya. The Hierarchical Timing Pair Model for Multirate DSP Applications. In *IEEE Transactions on Signal Processing*, volume 52, no. 5, May 2004.

[4] A. Das, W. Dally, and P. Mattson. Compiling for Stream Processing. In *Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2006.

[5] L. de Moura and B. Dutertre. Yices 1.0: An Efficient SMT Solver. In *The Satisfiability Modulo Theories Competition (SMT-COMP)*, August 2006.

[6] W. Du, R. Ferreira, and G. Agrawal. Compiler Support for Exploiting Coarse-Grained Pipelined Parallelism. In *Supercomputing Conference (SC)*, Nov. 2003.

[7] A. Eichenberger and E. Davidson. Efficient Formulation For Optimal Modulo Schedulers. In *Proc. of Programming Language Design and Implementation*, pages 194–205, June 1997.

[8] J. Glossner, E. Hokenek, and M. Moudgill. The Sandbridge Sandblaster Communications Processor. In *3rd Workshop on Application Specific Processors*, pages 53–58, Sept. 2004.

[9] M. Gordon et al. A Stream Compiler for Communication-Exposed Architecture. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2004.

[10] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2006.

[11] M. Hall et al. Maximizing Multiprocessor Performance with the SUIF Compiler. In *IEEE Computer*, Dec. 1996.

[12] H. Holma and A. Toskala. *WCDMA for UMTS: Radio Access For Third Generation Mobile Communications.* John Wiley and Sons, LTD, New York, New York, 2001.

[13] G. Kahn. *The semantics of a simple language for parallel programming.* J.L. Rosenfeld, Ed. North-Holland Publishing Co., 1974.

[14] E. Lee and T. Park. Dataflow Process Networks. *Proc. IEEE*, pages 773–801, 83 1995.

[15] E. A. Lee. Overview of the Ptolemy Project. In *Technical Memorandum No. UCB/ERL M03/25, University of California, Berkeley*, July 2003.

[16] E. A. Lee and S. Ha. Scheduling Strategies for Multiprocessor Real-time DSP. In *Global Telecommunications Conference*, pages 1279–1283, Nov. 1989.

[17] Y. Lin et al. SODA: A Low-power Architecture For Software Radio. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, 2006.

[18] J. L. Pino, S. Bhattacharyya, and E. Lee. A Hierarchical Multiprocessor Scheduling System for DSP Applications. In *Twenty-Ninth Annual Asilomar Conference on Signals, Systems, and Computers*, Oct 1995.

[19] B. R. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelined Loops. In *Proc. of 27th Annual International Symposium on Microarchitecture*, pages 63–74, Nov. 1994.

[20] H. Rong et al. Single-Dimension Software Pipelining for Multi-Dimensional Loops. In *Proc. of the International Symposium on Code Generation and Optimization*, March 2004.

[21] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proc. of the 2002 International Conference on Compiler Construction*, June 2002.

[22] C. van Berkel et al. Vector Processing as an Enabler for Software-Defined Radio in Handsets From 3G+WLAN Onwards. In *Proc. 2004 Software Defined Radio Technical Conference*, Nov. 2004.

[23] S. wei Liao, Z. Du, G. Wu, and G.-Y. Lueh. Data and Computation Transformations for Brook Streaming Applications on Multiprocessors. In *International Symposium on Code Generation and Optimization(CGO)*, March 2006.