In Proceedings of the IEEE Workshop Signal Processing Systems, Beirut, Lebanon, October 2011.

VECTORIZATION AND MAPPING OF SOFTWARE DEFINED RADIO APPLICATIONS ON HETEROGENEOUS MULTI-PROCESSOR PLATFORMS

George F. Zaki[1], *William Plishker*[1], *Shuvra S. Bhattacharyya*[1], *Charles Clancy*[2], *John Kuykendall*[3]

[1]Department of Electrical and Computer Eng., University of Maryland – {gzaki, plishker, ssb}@umd.edu
[2]Bradley Department of Electrical and Computer Eng., Virginia Tech, Virginia, USA – tcc@vt.edu
[3]Laboratory for Telecommunications Sciences, College Park, Maryland, USA – jbk@ltsnet.net

ABSTRACT

A variety of multiprocessor architectures have proliferated even for off-the-shelf computing platforms. To improve performance and productivity for common heterogeneous systems, we have developed a workflow to generate efficient solutions. By starting with a formal description of an application and the mapping problem we are able to generate a range of designs that efficiently trade-of latency and throughput. In this approach, efficient utilization of SIMD cores is achieved by applying extensive block processing in conjunction with efficient mapping and scheduling. We demonstrate our approach through an integration into the GNU Radio environment for software defined radio system design.

Index Terms— Design Methodology, Software Defined Radio, Graphic Processor Unit, Multiprocessor Scheduling.

1. INTRODUCTION

As frequency gains for individual cores level off, compute intensive application domains are increasingly turning to multiprocessor computing to continue achieving performance gains. A variety of multiprocessor architectures have proliferated including graphics processors (GPUs), multicore general purpose processors (GPPs), tile architectures, and multiprocessor digital signal processors. Even for off-the-shelf computing platforms, a heterogeneous mix of multiprocessor devices is likely, including at least one GPU and a multiprocessor GPP. Fully utilizing such a multiprocessor setup requires the identification of appropriate parallelism in the application domain and the implementation of such parallelism efficiently on the targeted platform. Software defined radio (SDR) is well positioned to make use of these trends. While performance gains were traditionally derived from uniprocessor performance improvements, ample application parallelism exists to target multiprocessor devices.

However, achieving efficient multiprocessor mappings of SDR applications on heterogeneous platforms poses significant challenges. When deriving mappings onto heterogeneous multiprocessors, designers must in general balance computational loads, efficiently use the given processor types, and account for communication costs. In SDR, this problem is further complicated by multirate application descriptions, and the need to satisfy constraints on throughput.

When targeting a GPU or other SIMD platform, vectorization must also be considered. More vectorization tends to lead to higher utilization of the platform (and therefore higher throughput), but often at the expense of increased latency and buffer memory requirements. Also an accelerator typically requires significant latency to move data to or from the host processor, so sufficient data must be burst to the accelerator to amortize such overheads. Ideally, application designers would be simply presented with a Pareto curve of latency versus vectorization trade-offs so that an appropriate design point can be selected. However, vectorization generally influences the efficiency of a given mapping. Thus, to fully unlock the potential of heterogeneous multiprocessor platforms for SDR, an automated way of arriving at quality solutions is desirable.

For a design flow to generate such solutions, we begin with a formal description of an SDR application, which we extract from a GNU Radio [1] specification. This formal description in dataflow is an ideal starting point for dealing with multirate scheduling. We then use the application graph and schedule along with a vectorization value to construct a problem formulation that may be solved efficiently by a Mixed Linear Programming (MLP) engine that reduces the final mapping latency. By changing the vectorization level, we are able to generate a Pareto curve of vectorization and mapping results. We integrate this approach into the new workflow shown in Figure 1. To demonstrate the effectiveness of this approach, we take a mapping result determined by our approach and implement it in GNU Radio, and evaluate the performance compared to baseline results.

2. BACKGROUND

Dataflow graphs are widely used in the modeling of signal processing applications. A dataflow graph G consists of set of vertexes V and a set of edges E. The vertices or *actors* represent computational functions, and edges represent FIFO buffers that can hold data values, which are encapsulated as

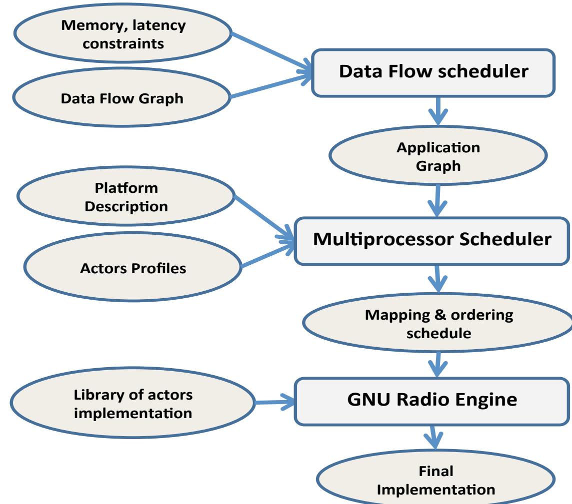

Fig. 1. Implemented Framework.

tokens. Depending on the application and the required level of model-based decomposition, actors may represent simple arithmetic operations, such as multipliers or more complex operations as turbo decoders.

A directed edge $e(v_1, v_2)$ in a dataflow graph is an ordered pair of a source actor $v_1 = src(e)$ and sink actor $v_2 = snk(e)$, where $v_1 \in V$ and $v_2 \in V$. When a vertex v executes or *fires*, it consumes zero or more tokens from each input edge and produces zero or more tokens on each output edge. Synchronous Data Flow (SDF) [2] is a specialized form of dataflow where for every edge $e \in E$, a fixed number of tokens is produced onto e every time $src(e)$ is invoked, and similarly, a fixed number of tokens is consumed from e every time $snk(e)$ is invoked. These fixed numbers are represented, respectively, by $prd(e)$ and $cns(e)$. Homogeneous Synchronous Data Flow (HSDF) is a restricted form of SDF where $prd(e) = cns(e) = 1$ for every edge e.

Given an SDF graph G, a *schedule* for the graph is a sequence of actor invocations. A *valid schedule* guarantees that every actor is fired at least once, there is no deadlock due to token underflow on any edge, and there is no net change in the number of tokens on any edge in the graph (i.e., the total number of tokens produced on each edge during the schedule is equal to the total number consumed from the edge). If a valid schedule exists for G, then we say that G is *consistent*. For each actor v in a consistent SDF graph, there is a unique *repetition count* $q(v)$, which gives the number of times that v must be executed in a minimal valid schedule (i.e., a valid schedule that involves a minimum number of actor firings). This minimal schedule executes a unit of execution that we refer to as one *iteration* of the given SDF graph. Furthermore, associated with any valid schedule S, there is a unique positive integer B, called the *blocking factor* of S, such that S invokes each actor v exactly $B \times q(v)$ times [2]. This operation is also known as *vectorization* of S.

In general, a consistent SDF graph can have many different valid schedules, and these schedules can differ widely in the associated trade-offs in terms of metrics such as latency, throughput, code size, and buffer memory requirements [3]. Figure 2 shows a typical SDF graph to model the mp-sched benchmark, which we describe later in Sec. 6. The repetition counts for this example are: $(SRC, 1)$, $(A11, 1)$, $(A21, 1)$, $(A12, 2)$, $(A22, 2)$, $(SNK, 6)$.

3. RELATED WORK AND CONTRIBUTIONS

Many models of computation have been suggested to describe software radio systems. In [4], the advantages and drawbacks of various models are investigated. In [5], a hierarchical dataflow programming approach is suggested to specify SDR graphs, and the Satisfiabily Modulo Theory is used to formulate the scheduling problem in order to increase system throughput subject to platform memory constraints. In [6], the authors present a multicore scheduler that maps SDF graphs to a tile based architecture. The mapping process is streamlined to avoid the derivation of equivalent HSDF graphs.

Various heuristics and mixed linear programming models have been suggested for scheduling task graphs on homogeneous and heterogeneous processors (e.g., see [7]). In these works, the problem formulations are developed to address different objective functions and target platforms for implementing the input application graphs. Vectorization for single-processor implementation of SDF graphs has been studied previously (e.g., see [8]). In [9] automatic "SIMDization" (conversion to a form that utilizes SIMD acceleration on the target processor) of streaming programs from a general purpose programing approach is proposed. A combination of SIMDization techniques with homogenous multiprocessor scheduling is also discussed.

In contrast with prior work, we begin with applications described in a domain specific environment (i.e., GNU Radio), and allow designers to use their existing optimized libraries alongside GPU accelerated library elements. We target platforms that consist of multiple GPP and GPU components, and systematically integrate SDF vectorization and inter-actor (task-level) parallel scheduling to optimize application throughput and latency on the targeted class of heterogeneous multiprocessor platforms. In the current GNU Radio engine, a strictly runtime multiprocessor scheduler is used to run applications through dynamic scheduling. However, for a wide range of SDR systems, offline profiling and analysis is possible, and more efficient scheduling solutions can be computed statically. To exploit such static scheduling opportunities, we provide an MLP formulation for the targeted multiprocessor scheduling problem. Our approach is restricted to acyclic SDF graphs, which can be used to represent a broad class of practical SDR applications and subsystems.

The primary contribution of this paper is a novel work-

flow for scheduling SDF graphs while taking into account actor execution times, efficient vectorization, and heterogeneous multiprocessor execution. This scheduling workflow is targeted carefully towards heterogeneous platforms that consist of GPPs and GPUs and applications described in a domain specific optimized language.

4. ARCHITECTURE AWARE SCHEDULING

4.1. From Application Model To Block Processing DAGs

A major set of primitive signal processing blocks in GNU Radio can be characterized as Synchronous Blocks: where the ratios between the consumption and production rates of edges are fixed. These blocks can be easily mapped to SDF actors.

For platforms that consist of both GPPs and GPUs, different levels of parallelism can generally be exploited in order to improve throughput. First, a fine grain level of data parallelism can be applied by utilizing the SIMD cores available in GPUs and vector operation accelerators in GPPs (if available). This level can be exploited using vectorization. A more coarse grain form of task parallelism is applied by mapping parallel tasks of the application graph onto the available set multiple processors. Both forms of parallelism may generally be exploited more effectively when $B > 1$, where B is the blocking factor. Under such a scheduling approach, the latency for a single graph iteration may increase. However, the latency for a block of B successive graph iterations may be reduced significantly, which leads to an increase in throughput (in terms of executed graph iterations per unit time). Such a trade-off is favorable in many throughput-critical systems or in applications where the increased latency does not exceed the given latency constraint. In our workflow, we set the level of global vectorization before the mapping step to properly inform the multiprocessor scheduler of the vectorized running time of the actors in the application for each processor type. By doing so, we efficiently utilize the SIMD cores by simultaneously firing multiple graph iterations. Therefore the basic multiprocessor scheduler objective is set to minimize the overall latency L_B of B graph iterations, which provides an optimized graph execution throughput of N/L_B graph iterations per unit time. Here B is a parameter than can be changed flexibly in our framework to help explore the scheduling design space.

By applying actor-level vectorization (also referred to as *block processing*) [8] , we can process the maximum possible number of tokens per actor execution. For an SDF graph, this objective can be achieved by using a *flat schedule* of the input graph [8] . A flat schedule can be generated by deriving a topological sort, and invoking every actor v a number of times equal to $B \times q(v)$. While flat schedules have the potential to improve processor utilization and throughput, such schedules generally suffer from high memory usage. However, in this paper, our objective is to increase the utilization of SIMD cores and furthermore, the available memory on a typical GPU is not a constraint for the class of SDR applications that we are targeting. Given an acyclic SDF application graph G, our scheduling approach first generates a directed acyclic graph (DAG), which we call a *block processing DAG* (*BPDAG*) T. T is isomorphic to G, meaning that the sets of vertices and edges are in one-to-one correspondence with one another. Each vertex t in T represents a vectorized version of a specific vertex v in G with some vectorization factor k (i.e., t represents k successive invocations of v). We refer to each vertex in a BPDAG as a *task*.

The BPDAG is sent to the core of our multiprocessor scheduling engine to perform task mapping (assignment of tasks to cores) and ordering (ordering of tasks assigned to the same core). Figure 2 shows an example of transforming the mp-sched SDF example to its corresponding BPDAG for a blocking factor of 10. This blocking factor generates the vectorization factor as defined in section 2 and is used to derive each task in the BPDAG.

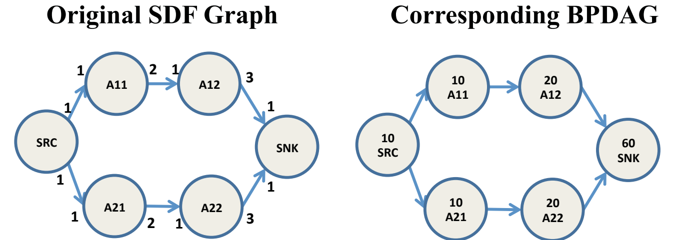

Fig. 2. Example of an SDF graph for the mp-sched benchmark and its corresponding BPDAG.

4.2. Architecture Model

In typical SDR platforms, the processing is executed by multiple heterogeneous processors that are suitable for different actor operations. Actors that perform control functions require complex pipelines and branch prediction units. GPPs (e.g., Intel quad cores) are usually suitable for these actors. Another relevant type of processor is the Single Instruction Multiple Thread (SIMT) type (e.g., NVIDIA GPUs). These processors have less sophisticated cores that are able to process individual functions on different data sets (e.g., symbol mapping and coding). Many physical layer actors require this kind of data parallelism, and GPUs often exhibit good performance for such actors. The target platform that we consider consists of a multi-core GPP, possibly with one or more SIMD accelerators (e.g., SSE extensions in Intel cores) accompanied with one or more GPUs. All of the processors are assumed to be connected with an all-to-all communication medium. If two dependent actors are allocated on the same processor, data movement will take place using shared memory at zero cost; otherwise, communication occurs across a contention-based communication medium (e.g., PCI bus).

4.3. Multiprocessor Scheduling Problem Formulation

The input to the multiprocessor scheduler consists of the following items.

a — Architecture description: The platform is described by a set P of processors and a set β of communication buses.

b — Application description: The application model (input BPDAG) consists of a set T of tasks, and edges E.

c — Dependency descriptions: Dataflow dependencies are defined by the src and snk functions described in sec 2.

d — Task and edge profiles: The task and edge execution times are obtained by simulating the tasks (edges) on different processors (communication media). These profiles are described by two functions: RTP($t \in T, p \in P$) $\rightarrow R$ defines the execution time of task t on processor p, and REB($e \in E, b \in \beta$) $\rightarrow R$ defines the execution time of edge e on bus b. Here, R is the set of positive real numbers.

e — Dependency analysis: Task t_1 is said to be *dependent* on task t_2 if there is a path that starts at t_1 and ends at t_2. If no such path exists between t_1 and t_2, then they are called *parallel* tasks. A similar concept can be applied to edges.

The input summarized in items *a-e* above is sent to the multiprocessor scheduler in order to perform the operations of mapping and ordering.

5. MULTIPROCESSOR MLP SCHEDULER

The problem description in sec 4.3 can be solved using available heuristics and optimal schedulers. As offline analysis is suggested to schedule static applications , a mixed linear programming (MLP) heterogeneous multiprocessor scheduler is proposed in order to find efficient solutions. The MLP scheduler consists of a set of equalities and inequalities that describe the application and architecture graphs, solution variables, constraints and objective.

5.1. Basic Variables

The basic MLP variables in our formulation are as follows.

- *Mapping variables:* $\forall t \in T$ and $\forall p \in P$, $XT[t,p] = 1$ if task t is assigned to processor p, and $XT[t,p] = 0$ otherwise. Similarly, $\forall e \in E$ and $b \in \beta$, $XE[e,b] = 1$ if edge e is assigned to bus b, and $XE[e,b] = 0$ otherwise.
- *Ordering variables:* $\forall$ parallel tasks t_1 and t_2 that are assigned to the same processor, $YT[t_1,t_2] = 1$ if t_1 is scheduled to run before t_2, and $YT[t_1,t_2] = 0$ if t_1 is scheduled to run after t_2. A similar formulation is applied for parallel edges.
- *Actual running time:* $\forall t \in T, RT[t]$ is the actual (platform-dependent) execution time of the task t depending on its mapping. Similarly, $\forall e \in E$, $RE[e]$ is the actual token transfer time for the edge e.
- *Start time:* $\forall t \in T, ST[t]$ is the start time for execution of task t. $\forall e \in E$, $SE[e]$ is start time of data transfer across edge e. These variables will be controlled by the dependencies expressed in the BPDAG and the ordering variables.

In this formulation, the basic variables (defined above) are used to derive a number of other variables. These derivations are carried out so that we can use linear equations to "detect" pairs of tasks that are assigned to the same processor. First we define the variables $ZTP[t_1,t_2,p]$, where $t_1 \in T, t_2 \in T, p \in P$, and $t_1 \neq t_2$. $ZTP[t_1,t_2,p]$ equals one if t_1 and t_2 are both assigned to p, and equals zero otherwise. Clearly, this variable depends on $XT[t_1,p]$ and $XT[t_2,p]$. This dependency can be linearized according to the following constraints:

- $ZTP[t_1,t_2,p] \geq XT[t_1,p] + XT[t_2,p] - 1$
- $ZTP[t_1,t_2,p] \leq XT[t_1,p]$
- $ZTP[t_1,t_2,p] \leq XT[t_2,p]$

Next, we define another set of variables $ZT[t_1,t_2]$, where $t_1 \in T, t_2 \in T$, $t_1 \neq t_2$. $ZT[t_1,t_2]$ equals one if the two tasks t_1 and t_2 are collocated. These variables can be easily derived by the following inequality:

$$ZT[t_1,t_2] \geq \sum_{p \in P} ZTP[t_1,t_2,p].$$

The derived variables ZT will be used in two cases. First, for collocated parallel tasks, these variables help to adjust the start times of tasks based on their ordering. Second, if a pair of tasks is connected by an edge, then these variables serve to make the corresponding edge transfer time equal to zero, which is appropriate since the communication occurs through processor shared memory.

5.2. Constraints

We use the following inequalities to formulate our targeted heterogeneous scheduling problem:

- *Assignment:* Every task (edge) is assigned to only one processor (communication medium):

  $$\forall t \in T,\ \sum_{p \in P} XT[t,p] = 1 \text{ and } \forall e \in E,\ \sum_{b \in \beta} XE[e,b] = 1$$

- *Task running time:* $\forall t \in T, p \in P$

  $$RT[t] \geq XT[t,p] \times RTP[t,p]$$

- *Edge running time:* $\forall e \in E, b \in \beta$

  $$RE[e] \geq XE[e,b] \times REB[e,b] - K \times ZT[src(e), snk(e)]$$

  where K is a very large number. The second term in this inequality models the "edge zeroing process" (i.e., the process of setting an edge's token transfer time to zero) if the source and the sink tasks of the edge are assigned to the same processor.

- *Starting times for dependent tasks:* $\forall e \in E$

$$SE[e] \geq ST[src(e)] + RT[snk(e)]$$

$$ST[snk(e)] \geq SE[e] + RE(e)$$

These two equations guarantee the proper execution order of dependent tasks by also taking into consideration relevant edge execution times.

- *Starting times for parallel tasks:* Orderings for parallel tasks can be achieved using an adaptation of an equality from [10]: $\forall$ *parallel* tasks $t_1 \in T$, and $t_2 \in T, t_1 \neq t_2$:

$$ST[t_1] \geq ST[t_2]+RT[t_2]-K(1-YT[t_1,t_2])-K\times ZT[t_1,t_2]$$

$$ST[t_2] \geq ST[t_1]+RT[t_1]-K\times YT[t_1,t_2]-K\times ZT[t_1,t_2]$$

Note that the last term effectively disables these inequalities if the two tasks are not collocated

Finally, the objective function minimized is the total graph latency (makespan) M, which can be specified by:

$$\forall t \in T, M \geq ST[t] + RT[t]$$

6. EVALUATION

We have implemented the proposed workflow, multiprocessor scheduler, and GNU Radio integration. In this section we present experiments based on these implementations. We have implemented the scheduling framework in the GNU Radio package, and we have experimented with the framework using the mp-sched benchmark that we introduced in Section 2. This benchmark describes a flow graph that consists of a rectangular grid of FIR filters. The dimensions of this grid are parameterized by the *number of stages* ($STAGES$) and *number of pipelines* ($PIPES$). The total number of FIR filters is thus equal to $STAGES \times PIPES$. The SDF graph in Figure 2 shows an example of a 2x2 grid. In this evaluation, the number of filter taps equals 60. This benchmark represents a non-trivial problem for the multiprocessor scheduler as all actors in different pipes can be executed in parallel. In practical GNU Radio system design, user input eliminates a significant part of the solution space by restricting the allocation of some actors to specific processors.

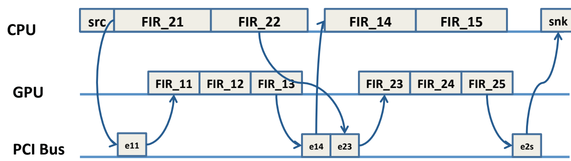

Fig. 3. Gantt chart for 2x5 mp-sched graph on 1 GPP and 1 GPU.

Table 1. Solver results for different mp-sched graphs.

Graph Size		Plat. Desc.		Lat	Solver
PIPES	STAGES	GPPs	GPUs	msec	hours
2	5	1	1	0.6	0.01
4	4	2	2	0.54	0.49
6	6	3	3	0.98	3.94
8	8	4	4	2.18	19.6

6.1. Test Setup and Workflow

The first stage of the workflow consists of the SDF scheduler, which reads the application SDF graph; calculates the repetition count $q(v)$ for every actor v; reads the global blocking factor B; and generates the corresponding BPDAG. Dependency analysis will be performed to set the required values associated with task dependencies. These operations are implemented using the DIF package [11]. In the second multiprocessor scheduler stage, the scheduler input is generated by setting the run-time of every actor and edge. This run-time will depend on the number of generated tokens per task invocation. The profiles of every actor for a given processor type are stored as tables that are indexed by the number of produced tokens. In this way, the tables are consistent with GNU Radio synchronous block descriptions. In this evaluation, this step is repeated for different blocking factors (B values). The MLP formulation is implemented using the GNU MathProg language [12]. This implementation consists of two parts: the problem description and data section. The MLP problem is solved using the IBM ILOG CPLEX optimizer.

6.2. Empirical and Solver Results

To evaluate our approach empirically, we selected a solution to implement within GNU Radio. Implementation details, based on a new GPU-oriented library for GNU Radio called *GRGPU*, can be found in [13]. Figure 3 shows the mapping and ordering solution for a 2x5 mp-sched graph running on a typical modern platform that consists of 1 GPP (Intel Xeon CPU 3GHz), 1 GPU (a NVidia GTX 260), and a PCI bus for a blocking factor $B = 2048$. According to the model, this is a 55% performance improvement over an all-GPP implementation and a 19% improvement over an all-GPU implementation. To validate the model result, we implemented this design within GNU Radio using profiling for latency per token and ensuring accuracy within 6 decimal places to the existing solution. With GRGPU, our solution provided a 39% performance improvement over our empirical results for an all-GPP solution, and and a 21% improvement over an all-GPU solution. For this level of vectorization ($B = 2048$), using both a GPU and GPP in the implementation provides the best results, as the model indicates.

Figure 4 shows a graph of latency per iteration for different vectorization levels. From the characteristic curves of the GPP and GPU implementations of the FIR actor, the GPU

is selectively used when I/O latency bound, but more heavily used when sufficient vectorization makes the problem compute bound for the GPU. Table 1 shows the solver running time and the latency (Lat) for different mp-sched graphs on various platforms. For the reported solver time, the *solution gap* ranges from 17% for the 4x4 graph to 67% for the 8x8 graph. By the solution gap, we mean the difference between the generally non-realizable results obtained from the real-valued solutions produced by the solver, and the practical results obtained from the corresponding integer valued solutions (derived by rounding the solver solutions). In these experiments, the MLP solver was executed on an Intel Core 2 Duo processor at 3 GHz.

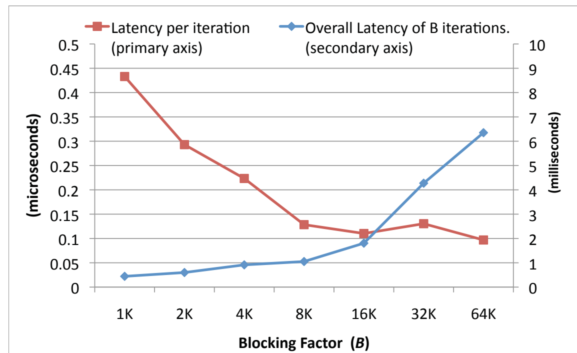

Fig. 4. Design space for a 2x5 mp-sched graph on 1 GPP and 1 GPU for different blocking factors.

7. CONCLUSION AND FUTURE WORK

Recent developments in SDR platforms involve heterogeneous multiprocessors, which target actors that require either complex architectures or data parallelism. In this paper, we have presented a workflow that takes advantage of formal models to describe the application, architecture and constraints, and produces efficient solutions for such platforms. In this approach, efficient utilization of SIMD cores is achieved by applying extensive block processing in conjunction with efficient mapping and scheduling. We have integrated this approach into the GNU Radio environment for SDR system design. Useful directions for future work include new graph transformation techniques for handling cyclic graphs, and handling of dynamic dataflow behaviors in addition to SDF graphs.

8. REFERENCES

[1] E. Blossom, "GNU radio: tools for exploring the radio frequency spectrum," *Linux Journal*, June 2004.

[2] E. A. Lee and D. G. Messerschmitt, "Synchronous dataflow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987.

[3] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, "Synthesis of embedded software from synchronous dataflow specifications," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 21, no. 2, pp. 151–166, June 1999.

[4] H.Berg, C. Brunelli, and U. Lucking, "Analyzing models of computation for software defined radio applications," in *Proc. IEEE International Symposium on System-on-Chip*, 2008, pp. 1–4.

[5] Y. Lin, M. Kudlur, S. Mahlke, and T. Mudge, "Hierarchical coarse-grained stream compilation for software defined radio," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, October 2007, pp. 115–124.

[6] S. Stuijk, T. Basten, M. C. W. Geilen, and H. Corporaal, "Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs," in *Proceedings of the annual Design Automation Conference*, June 2007, pp. 777–782.

[7] R. Niemann and P. Marwedel, "Hardware/software partitioning using integer programming," in *Proceedings of the European Design and Test Conference*, 1996, pp. 473–479.

[8] S. Ritz, M. Pankert, and H. Meyr, "High level software synthesis for signal processing systems," in *Proceedings of the International Conference on Application Specific Array Processors*, August 1992.

[9] A. Hormati, Y. Choi, M. Woh, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke, "Macross: Macro-simdization of streaming applications," in *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Opreating Systems*, March 2010.

[10] D. Applegate and W. Cook, "A computational study of the job-shop scheduling problem," *ORSA Journal On Computing*, vol. 3, no. 2, pp. 149–156, Spring 1991.

[11] C. Hsu, M. Ko, and S. S. Bhattacharyya, "Software synthesis from the dataflow interchange format," in *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, Dallas, Texas, September 2005, pp. 37–49.

[12] A. Makhorin, "Modeling language gnu mathprog, language reference," December 2008.

[13] W. Plishker, G. F. Zaki, S. S. Bhattacharyya, C. Clancy, and J. Kuykendall, "Applying graphics processor acceleration in a software defined radio prototyping environments," in *Proceedings of the International Symposium on Rapid System Prototyping*, May 2011.