Annales Univ. Sci. Budapest., Sect. Comp. 37 (2012) 311–338

DECLARATIVE SCHEDULING OF DATAFLOW NETWORKS

Gábor Páli

(Budapest, Hungary – Cluj-Napoca, Romania)

Communicated by Zoltán Horváth

(Received January 15, 2012; revised March 5, 2012;
accepted March 10, 2012)

Abstract. It is common for domain-specific applications to be supported with a specialized model of computation. In the domain of digital signal processing, dataflow networks are commonly employed for describing such systems, and therefore to specify a way of execution. We have created a model using a pure functional programming language, Haskell, to capture such applications on a higher level that may be used to generate programs. However, the run-time performance of the resulting code does not yet meet the performance requirements of the field. We believe that the situation may be improved by providing tools for the application programmer to control how the program in question is scheduled and executed without worrying about with the low-level and error-prone details too much.

1. Introduction

As part of our previous work on design and implementation of a coordination language framework for embedded domain-specific languages [12], we have briefly touched the issue of executing specialized task graphs [1]. In our model,

Key words and phrases: Operating systems, functional programming, Haskell, domain-specific languages, scheduling.
2010 Mathematics Subject Classification: 97P50.
1998 CR Categories and Descriptors: D.4.7.
The Project is supported by KMOP-2008.1.1.2-08/1-2008-0002, POSDRU/6/1.5/S/3-2008.

we created a set of workers (executors) dynamically pick and execute domain-specific-language programs (tasks) as part of a dataflow graph. The number of workers matches the number of processing units of the given hardware in the ideal case. That latter may contribute to lowering the expectations from the supporting run-time environment, eventually making the compiled graphs run standalone on top of the bare metal. This is going to be described in nutshell in Section 2.

However, during the preliminary performance tests, we have observed that our first naïve stab at scheduling execution of graphs does not scale well for multiple workers – which is not surprising as relations between tasks induced by data dependencies in the dataflow graph are not enforced to be respected. To earn guarantees for that, tasks may be partitioned into pools, though the optimal organization of pools is hard to achieve automatically. It will be explained in details in Section 3.

A sudden insipiration of advantages of embedded domain-specific functional languages mixed with the need for heuristics from the programmer constructing the application that the dataflow graph represents leads us to the concept of declarative scheduling. Declarative scheduling allows the implementation of domain-specific scheduling constraints that helps to abstract away from the low-level scheduling details and rather focuses on the protocol implementation itself. We will continue with its introduction in Section 4.

Since we are discussing a concept important to the field of operating systems, it is beneficial to explain how to translate the featured high-level descriptions to a target language that is closer to the hardware, and what kind of run-time support is required for them in practice. This is going to be illustrated with the C programming language in Section 5. In relation to this, in order to verify and demonstrate the advantages of our approach, we implemented some measurements on performance and the code size in Section 6. Therefore, the reader may have grounds to objectively judge the benefits.

Scheduling is still a research active topic that will be studied in details and compared to our work in Section 7. Recent research work done on the topic in the context of functional programming, databases and cloud computing supports the idea that gives us a motivation to evaluate our concept in those settings too.

2. Our dataflow model

Complex software applications, like operating systems, usually consist of a set of routines that are combined and controlled from upper layers as basic building blocks. With the growth of the application, it becomes a challenge to keep up its maintainability and reliability. In our view, if we abstract away

from the platform-dependent details by modelling the components and their relationship on a higher level it results in a manageable design. We used a pure functional programming language, Haskell, together with the technique of language embedding to propose a solution achieving that goal. In this design, the aforementioned routines are described as programs of dedicated domain-specific languages, and there we extended the picture by adding a glue language, called `Flow` on top of them for orchestration.

A prime example of that is a simple audio processing application which is tailored from domain-specific programs written in the language for the domain, called Feldspar [3]. The application reads digitalized stereo sound signals from a sound device, applies an effect to it, then writes the result back to the device. As today's hardware usually supports concurrent execution of applications, our goal would be to parallelize this processing by splitting the input signal by (left and right) channels. Processing of channel data then may be split further into smaller steps: first we calculate the spectrum of the signal, then the effect, and finally, it is converted back to a waveform. The scheme of the application is shown in Figure 1.

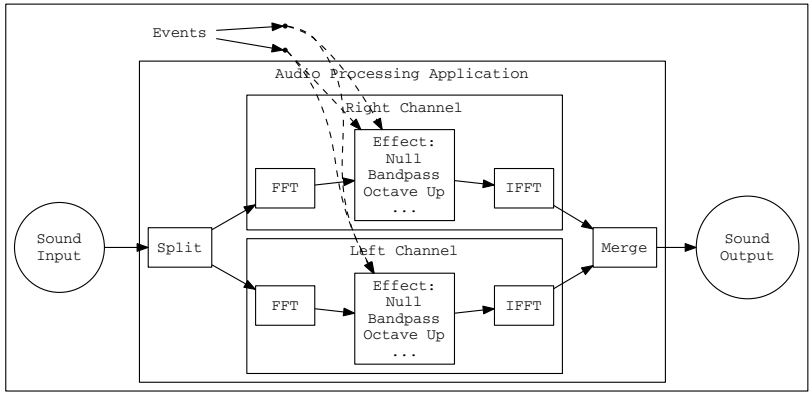


Figure 1. An overview of a sample audio processing application

There can be also seen that the application works with events, control instructions from the outside world to regulate its run-time behavior. Specifically, the application supports switching between a predefined set of sound effects (bandpass filtering, raising the pitch of sound by one octave), or set the parameters for the effects (the lower and the upper bound for bandpass filtering) dynamically.

Feldspar itself is a fine language for describing most of the application, though it is restricted to the domain. By looking back at Figure 1, we can note that while contents of the small boxes may be written in Feldspar, their combinations and the circle-shaped elements may not. Feldspar does not tell anything on how the programs should be run or organized into a working application. Thus another linguistic "layer" has to be added on the top of it.

2.1. The `Flow` language

To propose a solution for this problem, we have defined the `Flow` language that builds upon combinators to help to represent applications as data dependency graphs with computation nodes inside. Such graphs are called dataflow networks. Note that they have a domain-specific role as they express a loop in an event-driven system that constantly receives a stream of input and produces another stream of output as an answer. With `Flow`, it must be easy and convenient to construct applications on the shoulders of various domain-specific languages.

Domain-specific-language programs are used in their wrapped forms as nodes in the graph. Programs must have a special signature, they technically map an input (type `a`) to an output (type `b`). With wrapping, there may be parameters (type `c`) assigned to the programs. A set of run-time parameters associated with all the programs in the dataflow configuration is the global configuration. Those are the parameters that may be changed when the system is running. Note that this representation implicitly enables to compose events as changes in the subset of parameters. Besides the global configuration, there are nodes, called source nodes to receive input and sink nodes to produce output as a way to communicate with the environment.

The connection between nodes is typed, ensured by the `FlowType` type class, that guarantees the types used in the domain-specific languages can be always translated to a common (sum) type, `FT` to encapsulate data for transferring. It also establishes the method of communication between the languages themselves as each domain-specific language does not have to cross the boundaries of the node.

By using the `Flow` language, the application in Figure 1 can be written in a very succinct manner as follows.

```
audioproc = split --< (processLeft, processRight) >-- merge
  where
    processLeft   = fft --> effectLeft  --> ifft
    processRight  = fft --> effectRight --> ifft
```

where the functions `split`, `merge`, `fft`, and `ifft` are simple wrapped Feldspar

programs implementing the operations suggested by their names. In addition to that, the `effectLeft` and `effectRight` nodes are the specialization of the following program.

```
effectFeld :: Data Index -> (Data Index, Data Index)
  -> DVector (Complex Float) -> DVector (Complex Float)
effectFeld e bpp =
  switch e (noneFeld,bandpassFeld bpp,octaveUpFeld)
```

where `switch` implements switchable routing between certain Feldspar programs based on the value of index.

2.2. Decomposition of `Flow` graphs

Each dataflow network can be translated further down to an abstract program, an intermediate representation for the `Flow` compiler. Note that this representation still may be considered platform-independent. Abstract programs are built up from two main components, tasks and channels, expressed by the `Program` type. Note that the `FlowT` type here is used to denote the actual type of the `FT`-type elements communicated in the channels.

```
type ChanT = FlowT
type Program a b c t = ([Task c t],[ChanT])
```

A task represents a run-to-completion operation to be executed on a single processing unit without any interruption. The idea follows the hardware configuration and specifications usually featured in the domain, where there is only a limited support for preemption. Tasks communicate with each other via the channels. Tasks behave like closed expressions over their input, output channels and an actual state of the global configuration.

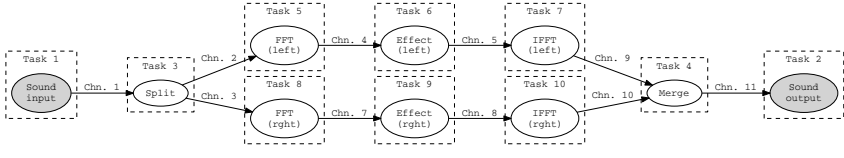


Figure 2. A potential decomposition of the audio processing application into tasks and channels

Tasks and channels are derived from the data dependency graph by decomposition: each node paired up with its incoming and outgoing edges as

channels is turned into a task. In Figure 2 such a decomposition is shown for the `audioproc` application.

Tasks are given by the `Task` algebraic data type. It has two type parameters: the first is the type of the global configuration (`c`), the second is the target language (`t`) – that generated code for, and the platform that provides the necessary abstractions to support running the given application. The exact definition is omitted for the sake of clarity.

3. Obstacles in efficient execution

After the data dependency graph has been decomposed into a set of tasks and a set of channels, it is ready to be executed – possibly on multiple processing units. A naïve solution would be to run each task on a different execution thread, but that would also require to spawn unbounded number of threads in general – which is a waste of resources, and tasks can easily outnumber the actual hardware execution units, forcing the run-time system implement time-shared multitasking. On the contrary, today's many- and multi-core hardware (that is typically featured in such domain-specific systems, e.g. Tilera boards) rather follows the trend of space-shared multitasking where the cost of context switching may be spared by completely dedicating a core (or more) to a given task for the entire time of its execution [16].

3.1. Task execution management

Hence our model relies on a run-time system that operates with a fixed number of threads. Such threads are considered "workers" that are free to assign ("grab") a task for execution for themselves from a pool. Because workers may pull in new tasks only when they have completed the previous one, the scheme offers sharing the load between the processing units dynamically, which is then incorporated as an implicit part of the model.

Execution of a task is as follows. Read the input channels, run the wrapped program with the gathered input information and the actual state of the global configuration, and finally place the results in the output channels. However, channels operate as FIFO (First-In-First-Out) queues that have fixed maximal lengths. Due to that, channels block writing when they are full, and block reading when they are empty. When the input data cannot be read or the output data cannot be written because any of its input and output channels is blocked, a task cannot be neither run despite it was selected for execution. Such "misses" in the scheduling may lead increased overhead costs where the

processor spends more time on selecting from the tasks than doing any actual work.

The abstract representation of channels in Haskell is as follows.

```
type Limit   = Int
type Channel = (Limit,[FT])
```

A task pool is given as a list of `Task`s waiting for execution.

```
type TaskPool c t = [Task c t]
```

There are two primary operations for task pools: `take` and `drop`. We are going to use the `STM` (Software Transactional Memory) [6] and the `IO` monads here to describe their abstract semantics in Haskell. The `STM` monad implements composable atomic memory transactions – all operations in an `atomically` block will be completed at one or retried when the shared resource becomes available again. Thus, the `atomically` function assumes proper synchronization for accessing shared variables. Shared data is described as `TVar`s (transactional variables) that can be read by `readTVar` and written by `writeTVar`.

The `take` operation gets a task from the pool. If there is no task found in the pool then it returns `Nothing`. Note that this is written as a single `STM`-block, which may be composed further.

```
take :: TVar (TaskPool c t) -> STM (Maybe (Task c t))
take pool = do
  t <- readTVar pool
  case t of
    (t:ts) -> do
      writeTVar pool ts
      return (Just t)
    _ -> return Nothing
```

The `drop` operation puts a completed task back to the pool. It shall always succeed as there must be enough place for the tasks in the pool.

```
drop :: TVar (TaskPool c t) -> Task c t -> STM ()
drop pool t = do
  ts <- readTVar pool
  writeTVar pool (ts ++ [t])
```

The introduced functions work with a program state (described by the `Program` type). It contains the input to be processed (list of values of type `a`), the generated output (list of values of type `b`) and a potentially infinite series of states of the global configuration (list of values of type `c`), and the state of the channels (as a list), respectively.

```
type State a b c t = ([a],[b],[c],[Channel])
```

Every worker receives references to a task pool and every worker accesses the same global variables, now they are summed up as a composition of tuples. First the worker tries to pick a task for itself: if succeeds, then it executes the selected task. Otherwise, the execution of the given worker is suspended (which is covered by the semantics of the `STM` monad hence not explicitly mentioned here) and re-try later to avoid busy waiting. The `updateStateWith` function is not detailed here – its purpose is to merge the values back to the global state that were changed locally.

```
workerStep :: (FlowType a, FlowType b)
  => TVar (TaskPool c t) -> TVar (State a b c) -> STM ()
workerStep pool state = do
  t <- take pool
  s <- readTVar state
  case t of
    Just task -> do
      updateStateWith state $ st task s
      drop pool task
    Nothing   -> return ()
```

Here the `workerStep` function describes a single step in the endless loop that a worker must implement.

```
worker :: (FlowType a, FlowType b)
  => TVar (TaskPool c t) -> TVar (State a b c) -> IO ()
worker pool state = forever . atomically $ workerStep pool state
```

Note that any task may be chosen from the pool. It would be the job of a scheduler to pick a task for a worker, however, it is not required as it can be expressed by `take` and `drop`. For now, let us suppose that `take` selects the first task from the pool (as defined above), while `drop` concatenates the completed task to the end of the pool (similarly).

That way we get the regular round-robin scheme, as presented in Figure 3.

In the case of multiple workers, the consequence of that strategy is not that straightforward as tasks may get reordered in the pool based on their time required for execution. This may cause problems in scheduling as if dependent tasks (that are connected to the given task via one of the channels) are executed in the wrong order, c.f. Figure 4. This could cause too many blocked tasks to be selected for execution, which results in a massive slowdown.

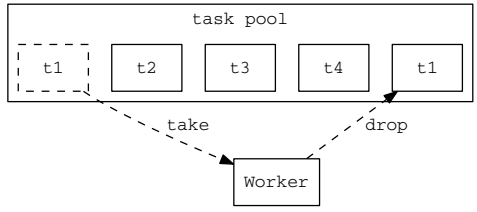


Figure 3. A single worker with a single pool

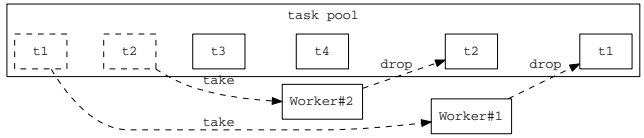


Figure 4. Multiple workers over a single pool
The order of elements may change during parallel execution

That can be relaxed by building up more task pools by taking the dependencies between tasks into consideration. It can be then implemented by partitioning the tasks into layers. Each layer contains tasks whose parallel execution would have bad effects on each other. Therefore, any of them may be selected for running in an arbitrary order, and the chances for blocking will be reduced to the minimum.

For example, consider the four tasks in Figure 3. Let us assume that they are connected in a shape of a diamond, that is t_2 and t_3 depend on t_1, while t_4 depends on t_2 and t_3. In this case, it would be better to create two pools: one for t_2 and t_3, and one for t_1 and t_4 (Figure 5). That way, workers will not violate the dependency ordering for the tasks.

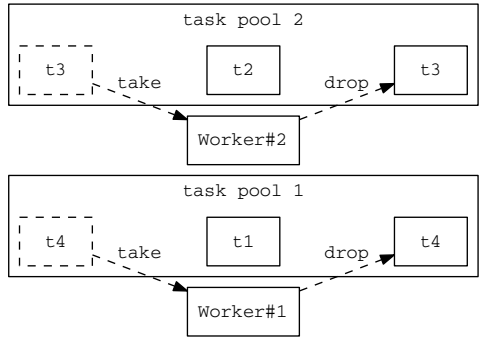


Figure 5. Multiple workers with multiple pools

3.2. Memory management

Apart from processor time, tasks may require a chunk of memory of a given size for their operation. The `Flow` language expects the applied domain-specific language implementations to be able to provide hints on how much memory is needed for running the wrapped program, called kernel here. In addition to this, there is a constant memory requirement for each task, e.g. the buffers to store data coming from or going to the channels.

Tasks get memory assigned when they are executed by a worker, and workers can only allocate memory on their start. That is, workers must have enough memory allocated to be able to run *any* task from the pool they are assigned to. As it implies, dynamic re-allocation of memory is strongly avoided here because they may degrade the performance (e.g. it may lead to fragmentation), and it requires some support for dynamic memory management from the run-time system. Assuming that sizes for the buffers and the kernel may be statically determined at compile time, it can be derived how much memory is actually needed for a given worker. It can be expressed by $M(\cdot)$ as follows.

$$M(w) = \max \left\{ \sum_{i=0}^{n_k} S(I(k,i)) + \sum_{j=0}^{m_k} S(O(k,o)) + S(H(k)) \;\middle|\; k \in K(w) \right\}$$

where w is the worker, k is a kernel from a set of kernel from that the worker my run ($K(w)$). The $S(\cdot)$ function determines the size for its parameter, and $H(\cdot)$ specifies the elements required for running the given kernel. Finally, $I(\cdot,\cdot)$ and $O(\cdot,\cdot)$ gives the nth input and output buffers for k, respectively.

Note that because of the *max* function above, certain kernels with high memory needs may cause wasting memory. For example, if the number of large-memory kernels is much more less than the number of small-memory ones, the workers may still have to allocate that much memory because of the large ones. That is, when memory requirements are not uniformly distributed among kernels, worker memory allocations would take the largest one. A solution for that would be also taking this into consideration when tasks are partitioned into pools.

4. How to be declarative

To address the problems sketched up here, we propose three ways to enhance the model with programmable scheduling, all of them are expressed in terms of an extension with a restricted language.

4.1. Marking boundaries for kernels

Note that because the kernel is the unit of execution, decomposing the `Flow` graph into tasks is crucial to the performance of the given program. Viz. that way the programmer gains some control over the granularity of the scheduling quanta as tasks are not interrupted. The programmer may choose to describe the application differently where the kernels are "fused" together in Feldspar, that is, all the intermediate data structures are removed as a result of the compilation. So, since there is no communication needed between the different processing units because of that, the serialization and deserialization of data may be simply optimized away. Moreover, as the `Flow` program enables the programmer to work on a higher level, that is, the result is usually compact and the details are left to a code generator, it is relatively easy to reorganize the kernels.

```
processFeld :: Data Index -> (Data Index, Data Index)
  -> DVector Float -> DVector Float
processFeld e bpp = ifftFeld . effectFeld e bpp . fftFeld
```

If respective specializations of `processFeld` were wrapped into kernels `left` and `right`, then our example sample can be rewritten as follows, with less but larger kernels.

```
audioproc' = split --< (left, right) >-- merge
```

With the possibility of describing task pool in a similar high-level way, we get the concept that we may call "declarative scheduling": the user specifies the way of how the program gets scheduled in terms of some high-level primitives without dealing with the low-level ones rather then adding them automatically as an increment of the combined use of the primitives.

4.2. Selectors

The basic idea is to mount a "selector" on the `take` operation. Here, it is expressed as a function over the elements in the task pool. It implies the following modification to the earlier definition of `take`.

```
take' :: TVar (TaskPool c t) -> TVar (State a b c t)
  -> Selector -> STM (Maybe (Task c t))
take' pool state f = do
  p <- readTVar pool
```

```
  case p of
    [] -> return Nothing
    xs -> do
      let (t,ts) = selectBy f xs state
      writeTVar pool ts
      return (Just t)
```

where the selector has the following type (given as a Generalized Abstract Data Type, GADT).

```
data Selector :: * where
  Next :: Selector
  By   :: Property -> Selector
```

Note that it has been applied together with the `selectBy` function which is used to specify the abstract semantics for each of the primitives above in Haskell.

```
selectBy :: Selector -> [Task c t] -> State a b c t
  -> (Task c t, [Task c t])
```

As shown above, the `Selector` type contains only an abstract description on how to pick the next task for execution. Brief explanation of the constructors is as follows, albeit note that there may be further similar functions introduced.

Next. Pick the next available task in the pool independently of the current state. This is the original behavior of the `task` function where the head of the task pool list was chosen.

```
selectBy Next pool _ = (head pool, tail pool)
```

By. Select a task from the pool based on a certain property, described by the `Property` type.

```
selectBy (By p) pool state =
  (snd $ head mins, map snd $ tail mins ++ misc)
  where
    (mins,misc) = partition (\(s,t) -> s == m) scoredPool
    m           = minimum (map fst scoredPool)
    score t     = (byProperty p state t, t)
    scoredPool  = map score pool
```

For the definition of `selectBy`, we used the `partition` function from `Data.List` standard Haskell module.

Note that we could use only the `By` combinator as a primitive operator for expressing selector functions. The distinction made here is rather categorical: the current position of a task in the pool (which the selection was based on) is not a property in the same sense as e.g. the length of its input queue. Expressing it as a property would also make the recognition of the given operator harder at code generation, hence that cannot be simplified to get an optimized translated version.

In the description above, the `byProperty` function was used to define semantics for the evaluating a certain property for tasks, and it has the following signature.

```
byProperty :: Property -> Task c t -> State a b c t -> Int
```

It is based on the `Property` type that is to list all the properties that may be calculated for the tasks. An example of such a property is the maximal input queue length.

```
qLen :: Property
qLen = QLen
```

which is represented by a data constructor function in the `Property` type.

```
QLen :: Property
```

whose semantics is expressed as follows.

```
byProperty QLen task (_,_,_,channels) =
  case task of
    TaskI _ _   -> 1
    TaskO _ chs ->
      maximum $ map (length . (channels !!)) chs
    TaskK _ (K _ _ chs _) ->
      maximum $ map (length . (channels !!)) chs
```

Here we use the received `State` value as the queueing information associated with each task that has to be retrieved from there with indexing the global channel table (`channels`).

Source nodes (`TaskI`) have to be handled specially: they do not have input channels and it is assumed that they can always provide data. Therefore a constant score is assigned to them. The selected constant here is `1`, which means the flow sips input only if there is no data inside to be forwarded, giving priority to data processing operations in this way.

Because we want to get the task with the maximal queue length, the calculated score has to be inverted. To implement that, simple property transformers can be introduced for the convenience of the users.

```
minimumOf :: Property -> Property
minimumOf = id

maximumOf :: Property -> Property
maximumOf = Invert
```

It requires the addition of the `Invert` constructor to the `Property` type, with the following semantics.

```
Invert :: Property -> Property

byProperty (Invert p) task = negate $ byProperty $ p task
```

So choosing a task by the maximal queue length can be written as follows.

```
maxQLen = maximumOf qLen
```

4.3. Creating task pools

Besides selection, it is also possible to control how task pools are organized. The purpose is to cover the problems of respecting data dependencies between tasks to minimize the chances for picking an already blocked task (as we have observed in Subsection 3.1) and uniform distribution of tasks by memory requirements (see Subsection 3.2).

Controlling organization of task pools requires the tasks to be identified somehow inside the application. Since the user has no exact knowledge on how tasks are dervied, there must be a tool for tagging flows provided instead.

```
type Pool = (Int, Selector)
```

As we will see through the section, task pools are described as a Cartesian product of an identifier (`Int`) and a selector (`Selector`). The former tells which task pool we are talking about at the given point of the flow, and the latter assigns a scheduler – as a function of the selector – to the pool.

In order to tag, pools have to be created first by the `createPool` function. There the user has the freedom to pick an identifier for the pool to be created. This will be exploited later.

```
createPool :: Int -> Pool
createPool id = (id, defaultSelector)
```

Here a default selector is assigned to each fresh pool, which is actually the previously characterized default behavior of the `take` function.

```
defaultSelector :: Selector
defaultSelector = Next
```

The `scheduleBy` function is used to override the semantics of the selector for one of the pools.

```
scheduleBy :: Selector -> Pool -> Pool
scheduleBy s (id,_) = (id,s)
```

We can only refer to nodes (but not tasks) in the high-level representation (see Section 2), so there we introduce an operator to implicitly annotate tasks to be created through their parent sub-flow. In our opinion, it is safe to do this as tasks are deterministically derived from the nodes (see Subsection 2.2).

```
infix 2 #=

(#=) :: (Backend t, FlowType a, FlowType b, FlowType c)
  => Pool -> Flow a b c t -> Flow a b c t
```

The `#=` operator annotates the given sub-flow of the `Flow` graph, assigning every derived task to the specified pool (together with all of its properties) during the decomposition. Such annotations may be then nested in a fashion similar to nesting mathematical sets.

```
pool1 #= flow1  --> (pool2 #= flow2 --> flow3) --> flow4
```

In the example above, tasks created from `flow1` and `flow4` are added to `pool1`, while tasks from `flow2` and `flow3` are added to `pool2`.

Note that pool tags are optional to use. If none of them is employed, the compiler shall select one of the built-in heuristics for organizing the tasks into pool and controlling the `take` and `drop` operations. It is because there is a default pool, `0`, which is automatically assigned to the entire flow to be compiled. Therefore, there is a "root pool" built up that contains all the tasks. Because of the style of nesting, this technically implies if there is no tag associated for the given sub-flow, then this is going to be the root pool by default. The same applies to the implicit source and sink nodes as well, i.e. they will be always in pool `0`.

Hence it may happen that we want to re-use an existing pool. We add the `usePool` function to express a partial pool label, where only the identifier is given.

```
usePool :: Int -> Pool
usePool = createPool
```

As the definition shows, it is actually a different name for `createPool` though.

Note that both the partitioning of tasks and the specification of selectors may be derived automatically. All that has to be done is to insert the pool tags by a given algorithm instead of doing it manually.

4.4. Example

By using the concepts we have introduced in the section, `audioproc` may be tagged in the following way to improve its performance.

```
[pool1,pool2] = map createPool [1,2]

audioproc = split --< (processLeft, processRight) >-- merge
  where
    processLeft  = pool1 #= fft --> effectLeft  --> ifft
    processRight = pool2 #= fft --> effectRight --> ifft
```

This way three independent task pools are created for the `Flow` program, grouping only the kernels that may be more or less executed independently of each other. This is presented in Figure 6.

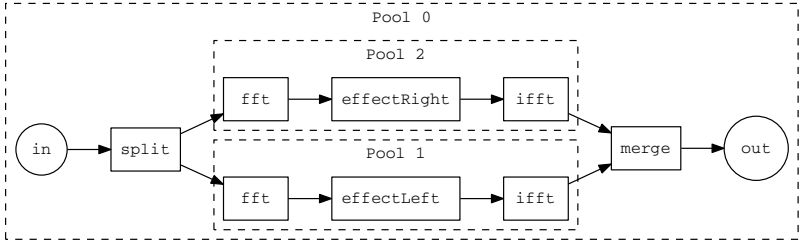


Figure 6. The audio processing application partitioned into three task pools

This may be improved further with addition of a simple dynamic scheduler, which picks a task with the longest input queue. We use the definition of `maxQLen` from Subsection 4.2.

```
pool = scheduleBy maxQLen $ usePool 0

audioproc = pool #=
  split --< (processLeft, processRight) >-- merge
    where
      processLeft  = fft --> effectLeft  --> ifft
      processRight = fft --> effectRight --> ifft
```

5. Code generation and run-time support

For the `Flow` model, code generation depends on the target language and platform. The approach does not restrict the choice of the target language – that is why we used Haskell to describe the algorithms in their abstract form earlier. The only requirement is there must be a mapping to that from the abstract program representation and the domain-specific-language kernels. For the rest of the document, we will use the C programming language as an example, unless explicitly noted otherwise.

For each target, there shall be a `compile` function implemented which generates a series of definitions (i.e. code, expressed as a Haskell list below) for the given target based on the name of the constructed flow. It must also contain an initial state for the global configuration.

```
compile :: (Backend t, FlowType a, FlowType b, FlowType c,
  Translation c (Types t)) => Flow a b c t -> c -> String
  -> [Definition t]
```

The `Translation` type class describes a relation between types, stating that there is a conversion from type `a` to type `b` if an instance for `Translation a b` exists. It is used to express that the type presenting the global configuration (`c`) may be translated to the target language.

During code generation, each node is translated to the target language by invoking the associated domain-specific-language compiler and some outer glue code is added to fit it for the run-time environment of the dataflow network to be created. The goal is to maintain the mapping between the domain-specific types and the `Flow` types in the backend. Besides that, there may be tasks in

the network that does not contain any kernel, but responsible for generating input data or consuming output data. Their definition is not part of the high-level program, so the user has to write code for them on the target language.

In addition to that, there must be a *controller* function. Its purpose is manifold. It controls the life time of the system: before it is started, the necessary initialization routines are run, and after it is finished, the run-time system shuts itself down. The controller accesses the internals of the global configuration therefore, it may change the parameters of the programs in the network, which may be then used to generate events in run time. It can be considered a delegated `main()` function of the application.

The generated code requires some run-time support to run. Those are the primitives required for operating the model presented through the document: task pools, workers, and message queues. The implementation of such abstractions depends on the *platform* where we want to run the system. The amount of actual run-time support required from the given platform is a matter of deployment. The software stack under the `Flow` program may consist of a conventional operating system (e.g. a microkernel, a set of servers representing system services or any POSIX-compliant system), and run-time libraries for the target language, or it may be run directly on top of bare metal.

Here we briefly present a sample API for any POSIX-compatible operating system.

```
struct worker_t;
struct taskp_t;
struct configuration;
struct taskinfo_t;
struct msgq_t;

typedef void (*task_ptr)(void *, struct configuration *);
```

The previously listed concepts are captured by the following (opaque) types: `worker_t` is a worker, `taskp_t` is a task pool, and `configuration` is the global configuration. The `task_ptr` is only a technical type, which is a function pointer. It points to the code of the corresponding task to be called (from the worker) as a regular function with a chunk of memory and the global configuration. Tasks in the array are stored in a `taskinfo_t` structure, which contains information on each of them, e.g. address the task body or the addresses of the associated queues. The `msgq_t` structure holds all the run-time information needed for working with message queues.

```
struct msgq_t *       queue[QUEUE_COUNT];
struct taskinfo_t *   task[TASK_COUNT];
```

```
struct taskp_t *      task_pool[TASK_POOL_COUNT];
struct configuration  config;
```

The actual `main()` function is generated along those types for the `Flow` program. It uses some global variables to maintain a state for the message queues (`queue` with the length of `QUEUE_COUNT` that is equal to the number of channels in the abstract program), tasks (`task` with the length of `TASK_COUNT` that is equal to the number of tasks in the abstract program), a task pool array (`task_pool` with a size corresponding to the number of actual pools (`TASK_POOL_COUNT`) in the entire program), and a global configuration (`config`).

The associated prototypes in the run-time system are the following.

```
struct worker_t *
worker_create(int, size_t, struct taskp_t *, struct taskinfo_t *,
  struct configuration *, int);

void worker_destroy(struct worker_t *);
void worker_start(struct worker_t *);
void worker_stop(struct worker_t *);
```

We can create and destroy workers, and we can also start or stop them after creation. This helps to follow the standard thread pooling pattern, where first we create all workers and then they are started. If it is possible on the given platform, processor affinity is set for each of them, so it is not allowed to move threads between processors. The opposite happens on shutdown. In addition to that, it can be also specified which pool to use, how much memory is given to the worker to work with, and a value for suspending the execution when it cannot find a task to run in the pool.

```
struct msgq_t * msgq_create(size_t);
void            msgq_destroy(struct msgq_t *);
size_t          msgq_send(struct msgq_t *, void *, size_t);
size_t          msgq_recv(struct msgq_t *, void *, size_t);
int             msgq_empty(struct msgq_t *, size_t);
int             msgq_full(struct msgq_t *, size_t);
```

There can be message queues created and destroyed as well. We can send and receive data over the queues, they represent the channels. The message queues work with bytes without types. That is, we do not use typed channels as the wiring has already been checked for correctness in the upper layer. It is a direct translation of the `FT` type that shares the same purpose.

```
#define NO_TASK_FOUND (-1)

typedef int task_t;
typedef task_t (*task_selector)(struct taskp_t *);
typedef int (*task_property)(task_t);

struct taskp_t *
taskp_create(size_t, task_selector, task_property);

void   taskp_destroy(struct taskp_t *);
void   taskp_add(struct taskp_t *, task_t);
task_t taskp_take(struct taskp_t *);
void   taskp_drop(struct taskp_t *, task_t);
```

Task index used for pointing to tasks within a pool is represented as an integer, called `task_t` here. Like in the previous abstractions, we can create and destroy task pools, and add the indices of the assigned tasks to them by using `taskp_add()`, and get the next appropriate free index by calling `taskp_take()`. When there is no task to be chosen, a special value, `NO_TASK_FOUND` is returned. After the execution, the `task_drop()` function is called to return the given task to the pool. Note that there is no `taskp_remove()` operation as task pools should not be changed in run time and `taskp_add()` is used only to build up the pools.

When creating task pools, a selector (as `task_selector`, a type synonym for a function pointer) and a property (as `task_property`, a function pointer as well) optionally used for task selection is passed together with the size. The selector must be a valid pointer, while the property may be `NULL`.

```
int  task_selector_next(struct taskp_t *);
int  task_selector_by(struct taskp_t *);
```

Selector receives the task pool to be handled. For the `task_selector_by()` selector, it can access the configured property (passed at the invocation of `task_create()`) because it is stored as meta-information for the given pool.

```
int  task_property_qlen(task_t);
```

Properties receive the task index only, so they can get all the task-related information (via the `taskinfo_t` structure) from the task array. Here a sample prototype for the queue length property is given, and further similar function may be implemented in addition to that.

As mentioned earlier, task pools may be also represented as queues where the next task to be selected is on the top, so a “put” operation automatically

picks that. For the `Next` selector that would correspond to a FIFO queue, and for the `By` selector that would be a priority queue, where priorities are calculated by the given property or a multi-level feedback queue, where the tasks are moved between the levels in reflection to the changes in the scores. Through the implementation the latter would be complicated as the priority queue had to be reordered constantly as the generated scores are changing, and multi-level queues had to be maintained in the same fashion. For that very reason, we found it easier to implement the queues by functions. If the number of elements in the pools are kept low, they may be an equally good decision in our humble opinion.

6. Experimental results

To evaluate the proposed solution, we have implemented it through embedding `Flow`, `Selector`, and `Property` into Haskell and working out a C code generator for them, which are then supplemented to the Feldspar compiler. We followed the specifications of the C run-time system described in Section 5 to implement that over the POSIX threading library, `pthread`. We used the `audioproc` audio-processing application, mentioned in Section 2. The sources were measured by `sloccount(1)`[1] (version 2.26, default settings), where the difference shown in the cost of implementation is out of question: the combination of `Flow` and `Feldspar` version is below 200 lines – SLOC: 147, while the generated C code is above 1000 lines – SLOC: 1017. The size of the accompanying client code written in C is of SLOC 89, while the run-time system is of SLOC 367.

The performance evaluation was both done on a Dell 2950 server (2x4-core Intel Xeon L5320 at 1.86 GHz) running `FreeBSD/amd64 9-STABLE` (of Feb 26, 2012), using the default FreeBSD system C compiler (GCC 4.2.1) and a TILExpress-20G board (64-core Tilera TILE64) running `Linux/tile64 2.6.26.7-MDE-2.1.2.112814`, using a Tilera-enabled version of GCC. In both cases we compiled the C code with global compiler optimizations (`-O2`) enabled. Fortunately, both of them implement `pthread`, however some operating-system-specific extensions had to be used to set processor affinity for the threads.

We measured the per-second throughput in kilobytes of the constructed dataflow network to see how efficiently scheduled it is. Note that the goal of the application is to maintain a speed of about 172 KB/s as that is the required bandwidth of a 44.1-kHz 16-bit stereo signal. Hence we modified the original application by counting the received bytes to learn how much data can flow through the network if there are no such limitations at the end. That is, both

[1] `http://www.dwheeler.com/sloccount/`

the source and sink nodes have unlimited bandwidth. The data is transferred in 64-element chunks between the nodes.

Furthermore, we did not use more than 2 hardware threads because the application can be conveniently cut only into two: both Figure 7 and Figure 8 indicate that it is still enough to achieve a stable doubled throughput. Having hardware threads is also important as our future goal is to deploy such graphs without the invervention of the `pthread` library. The weak output experienced on the Tilera board is a consequence of the architecture, which features many parallel but weak processing units, i.e. it demands parallelization of everything possible, and currently Feldspar does not support code generation for those boards (e.g. parallelization of loops).

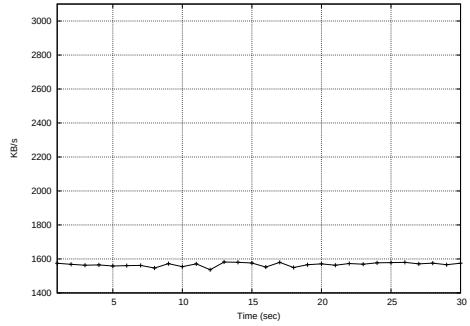
In the measurements, we compared three different variations of the `audioproc` example. The first version (Figure 7a, Figure 8a has a single pool with a single worker, i.e. that is a single-threaded application with round-robin scheduling between the tasks. It has a stable but slow throughput.

```
audioproc = split --< (processLeft, processRight) >-- merge
  where
    processLeft  = fft --> effectLeft  --> ifft
    processRight = fft --> effectRight --> ifft
```

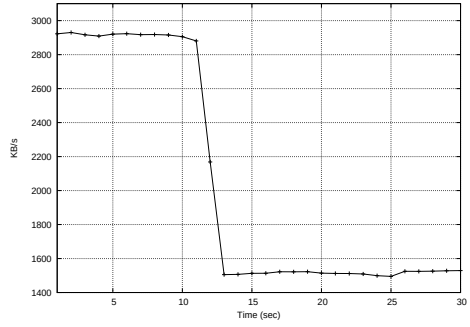
In the second variation (Figure 7b, Figure 8b),we do not change the high-level program but naïvely launch 2 worker threads (in the automatically generated code) to execute tasks from the same task pool. Albeit it starts up nicely, after a few seconds the performance dramatically collapses, even below the single-threaded case.

Thus an enhanced improved version is suggested to maintain both the high throughput and the stability. The results are displayed in Figure 7c, and Figure 8c. Note that while the previous version requires some implementation of mutual exclusion for the task pool, the latter does not as every worker has its own pool to work with.

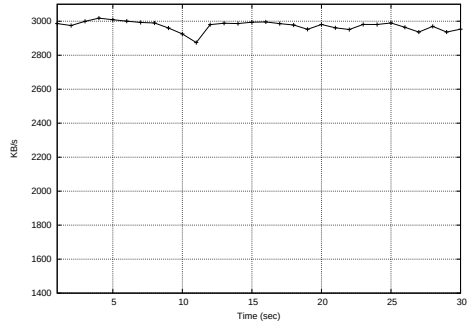
```
pool0 = scheduleBy maxQlen $ usePool 0
pool1 = scheduleBy maxQlen $ createPool 1

audioproc' = split' --< (processLeft, processRight) >-- merge'
  where
    split'       = pool1 #= split
    merge'       = pool0 #= merge
    processLeft  = pool0 #= fft --> effectLeft  --> ifft
    processRight = pool1 #= fft --> effectRight --> ifft
```



(a) A single pool with a single worker

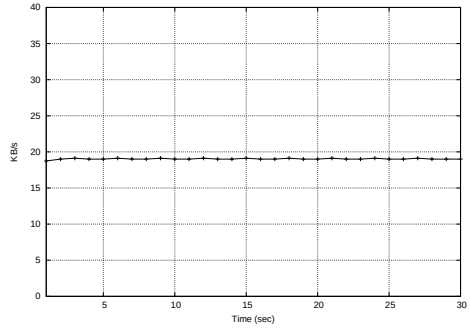


(b) A single pool with multiple (concurrent) workers

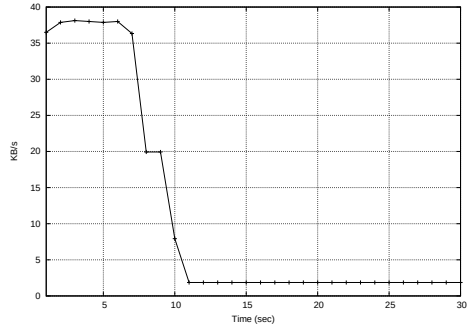


(c) Multiple (declaratively partitioned) pools with multiple workers

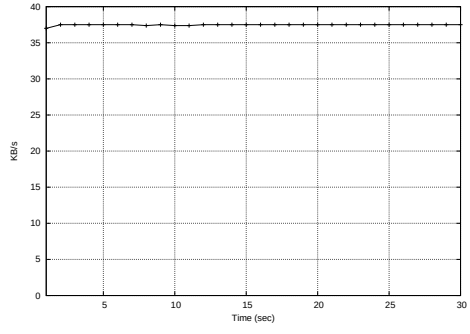
Figure 7. Performance results for running `audioproc` on a Dell 2950 with 2 hardware threads



(a) A single pool with a single worker



(b) A single pool with multiple (concurrent) workers



(c) Multiple (declaratively partitioned) pools with multiple workers

Figure 8. Performance results for running `audioproc` on a TILExpress-20G board with 2 hardware threads

7. Related work

Running and working with Haskell in the field of systems programming is a constant topic of the joint project of Galois, Inc. and the Portland State University, titled "High-Assurance Systems Programming" (HASP) [7]. In the past the researchers of this group have worked with the *House* operating system [5], which is implemented in Haskell. In Granuke's master's thesis [4] a branch of *House* is introduced, titled *Lighthouse*, which integrates the Lightweight Concurrency framework [8] for experimentation. As a result, the thesis features an extensible operating-system-level scheduler programmed in Haskell. It is similar to our thought as it tries to raise the process of development of schedulers to an abstract level, however, it still closely sticks to the traditional foundations. There the implemented scheduler is basically passive, that is, it may be rather considered a collection of routines written in Haskell. There is a simple interface defined but not for hints as we presented in the document. Because *House* originally lacks the notion of thread priority, *Lighthouse* utilizes that feature of the incorporated concurrency framework and extends handling of priorities in many different ways. The resulting extensible scheduler framework then facilitates easy implemenation of various scheduling policies. In our case though, no concept of priority is introduced as competing program codes are wrapped into cooperative tasks and the user is offered the opportunity to organize herself the optimal execution scheme for `Flow` programs.

Recent works of Marlow et al. [10, 11] show that expressing workflow systems in functional languages is still a hot research topic indeed. The `Par` monad is an extensive and generic tool to support parallel programming in a very efficient way. It does not do any I/O, hence it is considered pure and, therefore it can be used at many different places to describe similar (even dynamic) dataflow networks, where a scheduler can also be specified. But there the scheduler interface is not for general consumption, rather only provided as an "escape hatch" for relaxing certain situations. However, it uses many tricks inside (like `IORef`s) to make it work in Haskell and does not care about code generation. On the contrary, `Flow` concentrates on how to build automatically generated programs supported by a minimalistic run-time system. It then can be used for describing and compiling event-driven system programs for embedded hardware, without requiring a working Haskell compiler for that platform.

As an example of how to implement extensinble scheduling for a traditional microkernel-based system, L4 [9], Ruocco [13] proposes a user-level fine-grained adaptive real-time scheduling. This solution encourages developing systems that are capable of adapting themselves to a given problem by observing certain temporal properties of the execution. According to that paper, an optimal scheduling requires a two-way communication between the application and the

scheduler: the application forwards its decisions to the scheduler, while the scheduler respects them as much as it is possible. So there is an abstractions API implemented for L4 in C to offer the user a customizable scheduler.

In [14] and [15], Tilgner presents a research project that aims to define scheduling protocols in a declarative way for modern architectures, like web services or cloud computing. The reasoning behind his work is indeed understandable as cloud computing is typically a multi-computer system where good scheduling is cruical for the acceptable performance. In our case, the declarative nature is rather tied to the instrumentation of the run-time system proposed for our dataflow network model, and we use functional language techniques to offer a simple compositional solution. In [2], the term "declarative" is meant to characterize the data-oriented nature of the cloud computing solution of the authors. They use the Overlog language in conjunction with Haddoop MapReduce to work out declarative scheduling strategies for distributed systems.

8. Conclusion

In this paper, we presented some of the run-time aspects of our dataflow network modelling language, `Flow`, and proposed solutions to the experienced problems. The main contribution of the paper is to suggest the use functional-language combinators to build up a composable scheduling policy framework for a minimalistic dataflow language that relies on only a few abstraction to be implemented on each platform. Moreover, most of the details of the scheduling is implicitly implied by the model itself, helping the programmer to keep focusing on the relevant moments of the domain.

Finally, we would like to thank Tamás Kozsik and the anonymous reviewers for their valuable comments on the paper that greatly contributed to its development, and both the FreeBSD Project and the Tilera Corporation for providing professional hardware and software environment for testing and evaluation. This work is being supported by the Hungarian National Development Agency (KMOP-2008.1.1.2) and Programul Operaţional Sectional Dezvoltarea Resurselor Umane 2007–2013 (POSDRU/6/1.5/S/3-2008, Romania).

References

[1] **Agrawal, K., C.E. Leiserson and J. Sukha,** Executing task graphs using work-stealing, in: *Proc. of 24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010*, Atlanta, Georgia, USA, April 19–23, 2010.

[2] **Alvaro, P., T. Condie, N. Conway, K. Elmeelegy, J.M. Hellerstein and R. Scars,** Boom analytics: Exploring data-centric, declarative programming for the cloud, in: *Proc. of EuroSys 2010*, Paris, France, April 13–16, 2010.

[3] **Axelsson, E. et al.,** Feldspar: A domain-specific language for digital signal processing algorithms, in: *Proc. of the 8th ACM/IEEE International Conference on Formal Methods and Models for Codesign* (*MemoCode*), IEEE Computer Society, 2010.

[4] **Granuke, K.,** *Extensible Scheduling in a Haskell-Based Operating System*, Master's thesis, Portland State Universtity, 2010.

[5] **Hallgren, T. and M.P. Jones,** A principled approach to operating system construction in Haskell, in: *The 10th ACM SIGPLAN International Conference on Functional Programming*, 2005.

[6] **Harris, T., S. Marlow, S. Peyton-Jones and M. Herlihy,** Composable memory transactions, in: *Proc. of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (*PPoPP '05*), Chicago Illinois, June 15–17, 2005.

[7] The High-Assurance Systems Programming Project (Hasp). *Hasp Home Page*, `http://hasp.cs.pdx.edu/`, Date: 01-31-2012.

[8] **Li, P., S. Marlow, S. Peyton Jones and A. Tolmach,** Lightweight concurrency primitives for GHC, *Haskell Workshop 2007.*

[9] **Liedtke, J.,** On micro-kernel construction, in: *Proc. of the 15th ACM Symposium on Operating System Principles* (*SOSP*), pp. 237–250, 1995.

[10] **Marlow, S., R. Newton and S. Peyton Jones,** A monad for deterministic parallelism, in: *Haskell '11: Proc. of the 4th ACM SIGPLAN Symposium on Haskell*, Tokyo, Japan, ACM, 2011.

[11] **Marlow, S.,** Parallel and concurrent programming in Haskell, in: *Proc. of Central European Functional Programming Summer School* (*CEFP*), Eötvös Loránd University, Budapest, June 14–24, 2011 (to appear).

[12] **Páli, G.,** Extending little programs into big systems, in: *Proc. of the 4th Central European Functional Programming School* (*CEFP*), Budapest, Hungary, June 14 – 24, 2011 (to appear).

[13] **Ruocco, S.,** User-level fine-grained adaptive real-time scheduling via temporal reflection, in: *Proc. of the 27th IEEE Real-Time Systems Symposium* (*RTSS*), December 5–8, 2006, pp. 246–256.

[14] **Tilgner, C.,** Declarative scheduling in highly scalable systems, in: *Proc. of the 2010 EDBT/ICDT Workshops*, Lausanne, Switzerland, March 22–26, 2010.

[15] **Tilgner, C., B. Glavic, M.H. Böhlen and C. Kanne,** Smile: Enabling easy and fast development of domain-specific scheduling protocols, in: *Proc. of the 28th British National Conference on Databases* (*BNCOD*), Manchester, United Kingdom, July 12–14, 2011.

[16] **Wentzlaff, D. and A. Agarwal,** Factored operating systems (FOS): The case for a scalable operating system for multicores, *ACM SIGOPS Operating System Review: Special Issue on the Interfaction among the OS, Compilers, and Multicore Processors*, April 2009.

G. Páli
Eötvös Loránd University,
Department of Programming Languages and Compilers,
Faculty of Informatics
Budapest
Hungary

Babeş-Bolyai University,
Department of Computer Science,
Faculty of Mathematics and Computer Science
Cluj-Napoca
Romania
pgj@elte.hu