LIME: a future-proof programming model for multi-cores

Pjotr Kourzanov, Orlando Moreira, and Henk Sips

NXP Semiconductors Research, {Peter.Kourzanov,Orlando.Moreira}@nxp.com
Delft University of Technology, H.J.Sips@ewi.tudelft.nl

Abstract. The Less Is More (LIME) programming model addresses known programmability, compositionality, predictability, and scalability problems related to *parallel programming* in embedded systems of new as well as legacy code in streaming applications. With LIME, the *high-level functional* aspects of algorithm design and implementation are decoupled from the *low-level platform-specific* mechanisms pertaining to *communication* and *synchronization*. The integration of both in the end-product is assisted by a *tool-chain* that has complete access to the computations and has compile-time knowledge of *hardware-dependent performance aspects*. Rather than proposing intrusive modifications of a sequential language, LIME postulates *rules and restrictions* on how to express *algorithms* using standard C and *(de)compose* them using a simple XML schema for connecting components in a graph. In the paper, we describe the design rationales behind LIME and discuss its features in detail. We outline the LIME tool-chain, show how it interacts with analysis tools, and describe how multi-core back-ends are constructed. We illustrate this by showing a LIME implementation on a real-life parallel embedded platform for Software-Defined Radio (SDR) and an implementation on a commodity GPU platform.

Key words: multi- and many-cores, parallel, high-performance, embedded, programming model, real-time, streaming and data-flow models

1 Introduction & Problem Statement

The world is experiencing the multi-core revolution [10], and it will not be long before we enter the realm of *many-cores*. However, a definitive answer to the question of *programmability*, *compositionality*, *predictability* and performance/watt *scalability* of software running on such parallel hardware architectures is still not in sight. The complexity of parallel hardware is aggravated by the apparent and in many cases unneeded complexity in software, where layers upon layers at times tremendously complicate the task of the system designer.

A future-proof Parallel Programming Model (PPM) has to support exploitation of *variable-grain* parallelism - processor cores become ever *simpler* and *diverse* as their numbers rise. This is especially important in embedded computing, where performance/watt scalability largely drives modern heterogeneous Multi-Processor SoC (MPSoC) architectures [31]. Furthermore, parallelism needs to be

exploited on many levels: *task*, *data*, *memory*, and *instruction-level*. A shortcoming in one of these makes a PPM less applicable to some domains, forcing designers to create custom models. Finally, a PPM has to leverage the existing *legacy* code base. In both the embedded and High-Performance Computing (HPC) domains there are considerable amounts of proven algorithm libraries and a failure to approach these in an evolutionary manner would be very costly to amend.

There have been many attempts in the past to include concepts of parallel computing in traditional sequential programming languages. Section 2 gives an overview of existing approaches that address one or two of issues mentioned above, but unfortunately, not all simultaneously. In this paper, we show that our PPM for multi-cores does address all of the issues, and therefore enables system designers - architects *and* developers - to take one further step in making their software more *robust* and *flexible*, but still *efficient* and *scalable*.

The guiding principle behind LIME is based on the modern design practice of doing "More with Less". This way of design is distinguished by focusing on the elimination of the *complexity mismatch* between problem and solution. As a result, designs following this principle tend to exhibit *compositional* and *predictive* properties, which are very much desired in embedded and HPC applications. This can only be successful if a small number of *universal concepts* can be distilled from the problem domain and then *efficiently mapped* to system architectures and streaming application scenarios.

From an historical perspective, our approach can be related to the introduction of *high-level languages* in the 1950s. The goal then was to provide *abstractions* for data/memory access and control on top of lower-level machine languages. We think that in the era of multi-cores, a similar step is required to abstract from lower-level details of particular multi-core architecture. Designers should be able reason about systems and implement them on a conceptually higher level of *algorithms* and their *decompositions* that encapsulate *communication* and *synchronization*. Also, *performance* issues that arise when exploiting parallelism need to be tackled in a way that is orthogonal to implementation.

The paper is organized as follows: first, related work is covered in Section 2. The LIME PPM is introduced in Section 3 and its features are described. Two concrete applications of LIME are discussed in Section 4. Finally, conclusions and future work can be found in Section 5.

2 Related work

Parallel and concurrent programming have always been a difficult exercise. Following the tradition of focusing primarily on *efficiency* and *scalability*, each step in algorithm design and implementation is typically tightly coupled to a specific hardware architecture or to a specific PPM.

The usage of an API or the creation of libraries like MPI [5] or PVM [7] on embedded Real-Time (RT) platforms for streaming applications creates legacy problems on several levels. First, as the life-cycle of the application using one such library progresses (it is first created, then tested, and then maintained)

it becomes increasingly more and more difficult to make trade-offs, change the assumptions on the platform, or modify the performance contracts. Second, library APIs like POSIX threads (pthreads) that result from *standardization* tend to incorporate many different views, forcing conforming implementations to account for all views (*design-by-committee*). Also, these often lack abstractions that *shield* developers from readily-misused primitives. Another pitfall is abstracting too much or being too high-level (YAPI [29]), failing to address efficient usage in specialized embedded and HPC domains (*variable-grain* parallelism & *legacy*).

The industry in general and its embedded RT branch in particular is ever more reluctant to adopt new languages because of the legacy and inertia. Even high-profile research projects such as Ptolemy [14], that proposes extensive data-flow modeling & simulation environment, and StreamIt [8], that extends Java with data-flow constructs, fail to appeal to general-purpose and embedded community as a whole. If such PPMs find a good use it is usually in very specialized safety-critical and high-reliability domains, e.g., Esterel [2] & Lustre [4]. In the past, many different novel programming languages (Charm [1], Occam, Erlang, and many other research prototypes) have been tried to improve the programmability of parallel architectures. Also, intrusive modifications of existing languages that add new constructs, keywords etc. have been proposed (Parallel C [26], Cilk [18], Sieve [19], RapidMind [37] etc.).

More recently, a new wave of PPMs has resurged focusing on integrating the legacy sequential view with the new context of multi-cores, e.g., SMPSs [27] & CellSs [12]. Like LIME, these PPMs also advocate shielding of parallel complexity by constructs familiar to sequential programmers, making the task *life-cycle* (start/stop etc.) and *communication implicit*. In reality, however, too many concessions are made: like OpenMP (OMP) [6] these PPMs still require *explicit* control of task *scheduling* and *synchronization*. Rather than building new libraries strictly *on top* of a programming language some other recent PPMs try to find ways of expressing parallelism inside the type-system and standard library APIs such as C++ & STL. Although these look promising (see e.g., Ct [22] & TBB [30]), they still are very dependent on the designer to explicitly manage data- and task-level parallelism and carefully tune code using C++ features.

The SP@CE framework [40] also proposes a streaming programming model based on XML with embedded C algorithms. SP@CE targets only Series-Parallel graphs in the *soft real-time* Consumer Electronics (CE) domain, and proposes an *diversified* approach to control-flow comprising both C constructs and a *global event manager*. Unlike SP@CE, LIME approaches control-flow in a *unified* way, and addresses *ad-hoc* graphs in the *hard real-time* embedded, and HPC domains.

In summary, the industry is in an apparent need of standardized library APIs and PPMs. Although existing standards such as OMP [6] and MPI [5] have enough momentum, revolutionary General Purpose GPU (GPGPU) technologies like CUDA [36] and heterogeneous MPSoC systems (e.g., Cell/BE and some NXP offerings) challenge existing approaches on many different levels. Because of the features that it offers, we believe that LIME can serve as a *convergence point*

for shared- and distributed-memory programming models in general-purpose as well as in embedded and high-performance computing.

3 Programming model

In contrast to prior-art, LIME is neither an API (specific data-types, explicit functions or primitives are not prescribed), nor an intrusive modification of an old language (no extensions to C proposed), or a new Turing-complete language.

```
#include <unistd.h>
int MAIN() {
  const int buf[10]; int obuf[10];
  while (running) {
    if (!SelectIn(stdin,sizeof(buf)) ||
        !SelectOut(stdout,sizeof(obuf)))
      continue;
    Read(stdin,(int*)buf,sizeof(buf));
    compute(buf,obuf);
    Write(stdout,obuf,sizeof(obuf));
  }
}
```

```
#include LIME
void MAIN(const int buf[10],int obuf[10])
{
  compute(buf,obuf);
}
```

```
<edge type='fifo'>
 <from node id='stdin'/>
 <to node id='main' port id='buf'/></edge>
<edge type='fifo'>
 <from node id='main' port id='obuf'/>
 <to node id='stdout'/></edge>
```

Fig. 1. Example using pseudo-UNIX API (left) and its LIME equivalent (right).

LIME consists of two parts: components that contain algorithmic work, called *lime*s and a separate description of the dependencies between these *lime*s. These dependencies are contained in a *dependency graph* expressed in a *declarative* language called the Graph Exchange Format (GXF). As such, the communication and synchronization logic in the original C code, usually implemented using library APIs, is transformed to a graph description. GXF uses an Extensible Markup Language (XML) schema that has limited expressiveness, i.e. it does not support programming with flow of control/data constructs. An example is shown in Fig. 1, where the left part depicts the original C code and the right part depicts the *lime* (top right) and GXF description (bottom right). After transformation, the resulting *lime* only contains the algorithm, links to the GXF dependency expressions, and never uses platform-specific mechanisms directly.

In addition, LIME defines a set of *rules* that are enforced by the LIME toolchain on the GXF graphs as well as *restrictions* of a standard language such as C, see Subsection 3.2. Direct usage of this well-known sequential language is very important at this point because the embedded industry is known to posses a large volume of proven code and experience built around it. LIME flow supports an evolutionary path to multi- and many-cores by (1) extracting component models from C algorithms, (2) using models to map algorithms to tasks and (3) generating platform-specific mechanisms as well as (4) compilation and tuning of the algorithm for an architecture that contains many heterogeneous cores.

```
<id> = string
<type> = string
<size> = integer
<to> = <node>
<from> = <node>
```

```
<node> = <id> <type>?
         (<port>|<edge>|<node>)*
<port> = <id> <type>? <size>?
         <const>? <static>? <restrict>?
<edge> = <type> <from> <to>
```

Fig. 2. 11 tags of the GXF XML schema.

On another level, LIME mandates the use of a declarative GXF schema for the specification of the *synchronization structure*. Such structure can in fact contain either Data-Flow (DF), Control-Flow (CF), Series-Parallel (SP) patterns

or any combination of these. Although the current schema defines only 11 tags (see Fig. 2), the syntax is easily extendible. The GXF semantics include 4 rules that address graph *connectivity*, *hierarchy*, *composition* and *scoping*.

Although at this point in time our LIME prototype has a focus on SDR as an application [13], we argue that the basic concepts are also applicable to more dynamic forms of communication found in HPC and general-purpose computing. LIME supports parallelism on all levels, in fact:

1. Task-Level Parallelism (TLP) is *inherent*: software is *gradually* decomposed in a number of *components* of any required *grain*; each *lime* can (but is not required to) be a task.
2. Data-Level Parallelism (DLP) is *direct*: this is modeled naturally as *multi-rate* data flow and special *edge & port types*, see Subsection 3.2.
3. Memory-Level Parallelism (MLP) and Instruction-Level Parallelism (ILP) are *transparent*: LIME seamlessly integrates with existing C tool-chains.
4. *Evolutionary* approach to legacy software: LIME has a focus on C as a common programming language and avoids custom APIs.

Because not all forms of communication are analyzable, provisions are made as to bound, or relax expressiveness of the model depending on the level of guarantees that each particular application and use-case has to provide.

3.1 Compilation flow

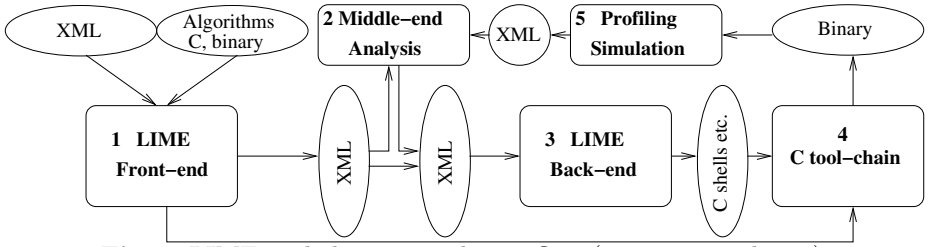

Fig. 3. LIME tool-chain compilation flow (`slimer` not shown).

The LIME tool-chain consists of the following engines:

1. *Front-End (FE) parsing* engine: responsible for converting C algorithms and GXF graphs into machine-readable format. Already at this stage the algorithms may be compiled, allowing 3rd parties to deliver binary components.
2. *Middle-End (ME) static analysis & scheduling* engine: responsible for static task *admission*, *mapping*, *grouping* and *scheduling*, see Subsection 3.3.
3. *Back-End (BE) code generation* engine: responsible for generation of platform-specific *shells*, OS *configuration*, and *startup* code, see Section 4.
4. *C tool-chain* that is used to compile generated and (optionally compile) algorithm code. This allows extra optimizations and automated performance tuning, if the algorithm source code is available to the platform integrator.
5. *Profiling & simulation* engine: provides *feed-back* to the ME.

A top-level compiler driver called `slimer` sequentially initiates the following compilation engines: $1 \mapsto 2 \mapsto 3 \mapsto 4 \mapsto 5$ (see Fig. 3).

The GXF language is also used to specify the distribution and connectivity of cores as well as the mapping of software nodes to hardware nodes. Together with the models extracted from the algorithms, this provides enough information for the automated generation of CF, DF and static task *life-cycle management* primitives. Most importantly for embedded RT systems, this arrangement supports *analysis* engines that tackle compositionality and predictability, where properties of the smallest units - *lime* algorithmic components - are used to predict the properties of a composition, such as a radio baseband modem application containing a graph of filters running on a Digital Signal Processor (DSP). The properties that are interesting for performance analysis include *timeliness*, memory/bus *footprint* as well as the usage of *resources* other than memory or cycles.

3.2 Syntax & Semantics

The current LIME syntax builds on existing concepts found in the ANSI C99 standard, as well as on GXF, which is loosely based on a semi-standard Graph Exchange Language (GXL) schema [3]. The same basic ideas, however, can also be applied to languages other than C and XML. For example, the graphs can be entered in a *visual* way and saved using the DOT format used in Graphviz [11], while C# could have been used to specify the algorithms. In this paper, however, we use C99 for *lime*s, GXF graphs in Figures 1 and 6, and DOT rendering elsewhere because of space constraints.

Basics. We will give an exposition of LIME using a typical working example from the data streaming domain using Synchronous DF (SDF) [32] with some extensions. This simple graph connects 3 components (*lime*s):

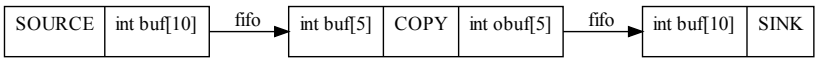

Fig. 4. Example graph (using DOT rendering).

1. `source` with one *out-port* (`buf`), used to inject data into the stream.
2. `copy` with one *in-port* (`buf`) and one *out-port* (`obuf`).
3. `sink` with one *in-port* (`buf`), used to verify this trivial computation.

These *lime*s are connected using edges typed "fifo". This refers to FIFO channels, although the exact implementation of these is of course hardware-dependent and subject to specific optimizations, see Subsection 3.3 and Section 4.

Fig. 5 depicts the complete listings using the C K&R syntax, which together with the graph in Fig. 4 provides enough information for `slimer` to generate actual platform-specific *shells*, the Operating System (OS) *configuration*, and

```
#include LIME

void PROCESS(buf)
  int buf[10];
{
  for(int i=0; i< 10; i++)
    buf[i]=i;
}
```

```
#include LIME
void PROCESS(buf,obuf)
  const int buf[restrict 5];
  int obuf[restrict 5];
{
  for(int i=0; i< 10; i++)
    obuf[i]=buf[i];
}
```

```
#include LIME

void PROCESS(buf)
  const int buf[10];
{
  for(int i=0; i< 10; i++)
    assert(buf[i]==i);
}
```

Fig. 5. Algorithm sources for source, copy and sink (graph in Fig. 4).

startup code to run the streaming graph on a parallel platform. The following **syntactic** properties of the LIME can be directly observed from this simple application:

- no explicit communication & synchronization calls are present in the input. Instead, the data dependencies are *isolated* inside the C function declaration and then made *explicit* in the graph. Using SDF terms, the actor *signature* coincides with the C function signature. This is effectively used by `slimer` to *generate a platform-specific shell* for each component and to *hook-up* its ports to other objects in the platform, e.g., *pipes*, *sockets* or FIFO channels.
- data-dependencies, or *ports* are *out-ports* by default. *in-ports* are specified using the C `const` qualifier. This gives compiler protection against unwanted writes. Also, the incorrect use of ports w.r.t. the *direction* or *type* can be signalled early in the compilation flow.
- *data-rates* are explicit in the signature as *array size* specifiers. As a result, the use of *pointers* is obsolescent. Furthermore, static code analysis can be used to calculate optimal schedules, FIFO buffer sizes (see Subsection 3.3) and to validate robustness properties (e.g., *out-of-bounds* accesses).
- the C99 `restrict` keyword can be used to indicate to the compiler that the port's data is never *shared* with other ports, i.e., avoiding aliasing. See the paragraph on *instantiation* below for more details.
- in- and out-port *rates* do not have to match - an edge can be *multi-rate*, see Fig. 4. This is an important source of DLP in LIME. If a producer writes more data than a consumer can read, our tool-chain, under some conditions, can choose to create as many instances of the consumer as needed (fan-out).

There are 3 **semantic rules** associated with the LIME model of computation that focus on the *embedding* of parallel abstractions in standard C99 and avoid task *management chores* related to *life-cycle*, *scheduling*, *communication*, and *synchronization* that are inevitably specific to each particular architecture:

1. The C *function call* as task *activation* - all inputs are assumed to be **ready** (using e.g., a read-lock or an *acquire*) and enough output space is assumed to be **available** (using e.g., a write-lock or an acquire). All ports must be used in this function, otherwise the C compiler will generate a warning.
2. The C *function return* as task *de-activation* - all inputs and outputs are **flushed** (using e.g., an unlock, or a *release*) and can not be used by this component until the next activation. This is enforced by the C language -

parameters (ports) have function *scope*, thus there is no way a component can use them when it is not active.

3. A C function can not by default assume anything about the **order of activation** - given enough resources all *lime*s (could) execute concurrently, their global order fully determined by the *graph* and by the platform-specific BE.

Exact implementation of instructions that mandate a particular *release consistency* model (acquires & releases, memory flushes) is completely defined by a particular LIME BE that is used to compile the application. Because of this decoupling, the BE can choose to apply *double-buffering*, or *in-place* processing, depending on platform and/or application requirements. Similarly, the BE may opt to generate *blocking* vs. *non-blocking* primitives (e.g., see Fig. 11). The only invariant that is maintained by LIME for each *iteration* and each *port* of a component is that all input and output data buffers are *contiguous* and exactly the specified by *data-rate* amount of data is *directly accessible* via a pointer argument, allowing the C tool-chain to exploit ILP present in the algorithm.

Data-flow extensions. Although the basic model looks simple, the abstractions it uses are powerful enough to be stretched for more advanced features:

Cyclo-Static DF (CSDF) allows further fine-grained decomposition of components into sub-components each having its own dependencies. This supports *late-acquire* and *early-release* optimization schemes as well as some level of encapsulation [15]. CSDF is expressed in LIME by simply specifying several *lime*s in one component and defining a local *static schedule* to order them. As depicted in Fig. 6, the signature of the super-component is the union of all sub-component signatures. The BE engine can optimize unneeded *acquires* and *releases*.

```
void PROCESS1(buf, obuf1)
const int buf[restrict 10];
int obuf1[restrict 5];
{
  for(int i=0; i< 5; i++)
    obuf1[i]= buf[i];
}
```

```
void PROCESS2(buf, obuf2)
const int buf[restrict 10];
int obuf2[restrict 5];
{
  for(int i=0; i< 5; i++)
    obuf2[i]= buf[5+i];
}
```

```
/* Statically order nodes:
 * process1 -> process2 */
void (*SPLIT_SCHEDULE[])()=
{
[1]=  PROCESS1,
[2]=  PROCESS2
};
```

```
<node id='split'>
  <port id='buf' type='int'
    size=10 const restrict/>
  <port id='obuf1' type='int'
    size=5 restrict/>
  <port id='obuf2' type='int'
    size=5 restrict/>
```

```
<node id='process1'>
  <port id='buf' type='int'
    size=10 const restrict/>
  <port id='obuf1' type='int'
    size=5 restrict/>
</node>
```

```
<node id='process2'>
  <port id='buf' type='int'
    size=10 const restrict/>
  <port id='obuf2' type='int'
    size=5 restrict/>
</node>
</node>
```

Fig. 6. Example (non-strict) CSDF split component and its GXF representation.

Variable-rate DF allows ports to accept data with variable rates. Of course analysability of the resulting graph will depend on the variation range. Variable-rate DF is expressed within C99 in a straightforward fashion using the Variable-Length Array (VLA) parameters: **void process(const int size, const int buf[size]);** Ranges and individual size requirements can be specified by *enumeration* of all possibilities: **void process(const enum {ZERO=0, ONE=1} size,const int buf[size]);** specifies that **process** is activated *irrespective* of whether there is 1 **int** in the input, or *none*, allowing asynchronous activation.

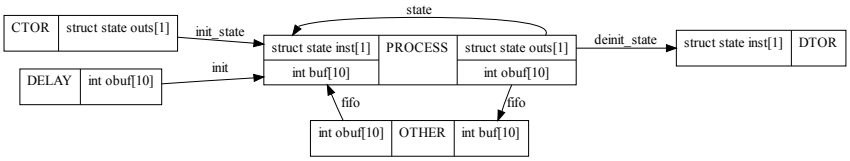

Fig. 7. Component having state (with a self-loop) and delay (loop via other)

Delays and instantiation. Any realistic PPM has to tackle practical issues associated to *multiple instantiation*, in addition to the complication of having *loop delays* as required by SDF analysis techniques. In LIME, both of these are modeled using *constructor* and *destructor* nodes. Such nodes are not different from regular DF nodes - what distinguishes them is the way they are *connected*. This is in fact specified in the graph by sub-typing edges as "init" or "deinit". Fig. 8 is an example of a component in which the state port does not use First-In First-Out (FIFO) protocol but rather maps to *memory* that is *shared* between component instance's activations. An edge specifies "state" type to indicate to LIME that the ports connected by this edge are *state* ports, as in Fig. 7. This information is needed by the tool-chain to calculate the state size per component, as syntactically state ports are not different from regular FIFO ports.

```
void CTOR(struct state outs[1]) {
  outs->param=123;
}
```

```
void DTOR(const struct state inst[1]) {
  dump(inst->param);
}
```

```
void DELAY(int obuf[10]) {
  memset(obuf,'\xFF',10*sizeof(int));
}
```

```
void PROCESS(buf,obuf,inst,outs)
  const int buf[restrict 10];
  int obuf[restrict 10];
  const struct state inst[1];
  struct state outs[1];
{
  for(int i=0; i< 10; i++)
    obuf[i]=buf[i]*inst->param;

  outs->param++;
}
```

Fig. 8. LIME specification of a state port & a delayed input port.

Instantiation is therefore achieved by specification of a constructor component that initializes the `process:inst` port through `ctor:outs` port, see Fig. 8. The tool-chain is responsible for generation of component state *allocation*, component *construction* and *garbage collection*, primitives.

Delays are specified by the constructor's port rate, see `delay:obuf` port in Fig. 8. In this case, the FIFO that is associated to the `process:buf` port is *pre-loaded* with the data generated by delay before the first activation of the process. This is needed to avoid deadlocks when the SDF graph contains loops.

Broadcast and reduction operations. These important concepts are supported in LIME by port name *expansion*. Collective operations are expressed with the `union` *type* for an out-port (broadcast) or an in-port (reduce), see Fig. 9. Note that the number of ports is not necessarily static, it can be *parameterized*,

as shown in `union collective_template`: the component may specify only one union *member*, indicating to LIME that it should instantiate as many ports as there are edge *endpoint* ports belonging to this component in the GXF graph.

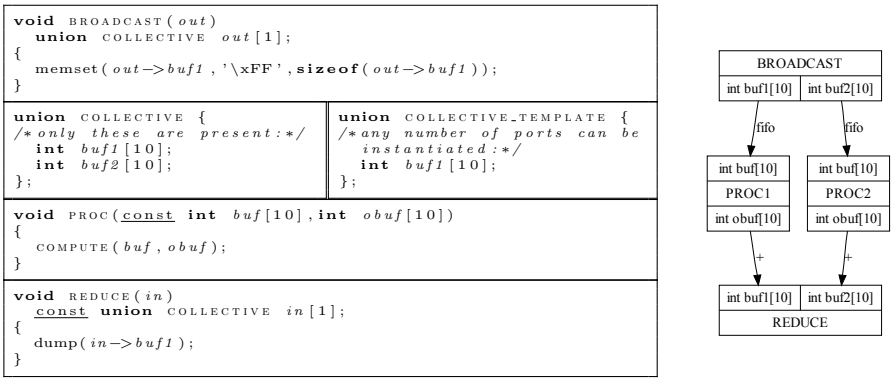

Fig. 9. Collective operations in LIME (+ reduction operator as the edge type).

Restricted Control-Flow. As the concept of *iteration* is inherent in LIME (all *lime*s are activated repeatedly until there is no input), only the concept of *asynchronous* activation and the concept of *conditional* activation needs extra attention. Both of these CF constructs are supported, the former by variable-rate DF (see above) and the latter by *variant record* port types (see Fig. 10). Similarly to *parameterized* collective template described above, `struct selective_template` containing an integer *tag* and a union with only one *member* (not shown in Fig. 10 because of space limitations) can be used to allow variability in the number of ports associated to conditional activation; the value of this parameter is then derived from the GXF graph description.

3.3 Applicable analysis models

Two different approaches to analysis are possible with LIME. The first one has roots in the classic SDF domain, which integrates very well with LIME. Another one originates in theory of SP graphs. Such graphs can also be expressed in LIME. Many embedded RT applications contain graphs that are SP (e.g., figures 4 and 9), making this another useful analysis methodology.

Data-Flow based. Different flavors of *data-flow modeling* allow for different degrees of temporal analysis, and for static allocation of resources such as *static scheduling* or *minimal buffer sizing*.

At one extreme, the Dynamic DF (DDF) model can express the full range of *Turing-complete programs*, but lacks many useful analytical properties. It may

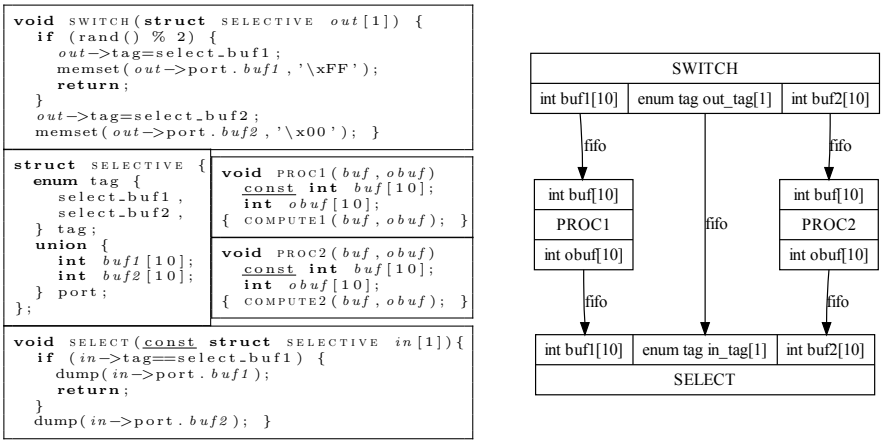

Fig. 10. Boolean DF (BDF) conditionals in LIME.

be impossible to verify for an arbitrary DDF graph that the *synchronization structure* it specifies is free of deadlocks. On the other hand, static data-flow variants such as SDF [33], Homogeneous Synchronous DF (HSDF) [38], CSDF [15] allow for powerful temporal analysis. This enables verification of execution properties such as *deadlock-freedom*, *latency* and *throughput* constraints [34] as well as determination of *maximum* achievable throughput [21] and *minimized* FIFO buffers [24] and even the generation of fully-static *rate-optimal schedules*.

It is clear that static models have limited expressiveness: they can only express applications that work with *fixed data rates*, i.e., the amount of data transferred per task activation (i.e., *actor firing*) is not dependent on input data. Because of this SDF models tend to be reserved for application domains where RT guarantees are required and where task activation is strictly data-driven. In between these two extremes there exist other variants.

Some restrictions of BDF [17], for instance, do not allow the generation of static schedules, but allow at least for generation of so-called *quasi-static* schedules [25]. Fig. 10 depicts a case with a boolean conditional. Besides BDF there are other related models (e.g., Integer DF), each with different properties.

All SDF models can be seen as a sub-set of DDF. To define what kind of DF model suits a particular application depends on firing rules for all of its actors. Since LIME makes the *firing rule* of each actor *explicit* in the function declaration, it is simple to detect by inspection of all function interfaces which DF properties are applicable, which enables the *correct-by-construction*, automatic generation of the analysis models that are needed for applying many of the *temporal analysis* and *resource allocation* strategies previously referred.

Series-Parallel based. Task scheduling and resource allocation algorithms that work with communication/synchronization graphs are related to a number of well-known problems in graph theory (*maximum independent set*, *graph col-*

oring). These are *NP-hard* in general, precluding their efficient *on-line* usage. Although static *off-line* schedulers & allocators can still be applied for some problem domains, many practical applications exhibit dynamic behavior.

One of the important results of graph theory is that for graphs that have limited *tree-width* k [39], i.e. for a partial k-trees (SP is a special case when $k = 2$) many of these algorithms have polynomial complexity [16]. Calculating k for any given graph, however, still is an *NP-complete* problem [9]. Most embedded RT applications (with some notable exceptions) contain algorithmic kernels that have SP synchronization structure [40]. Even if the structure is not strictly SP, it can be reduced to SP with bounded overhead [23].

LIME supports SP analysis frameworks by isolating *algorithm dependencies* in C modules and making the *algorithm decomposition* explicit in the GXF graph. The *off-line* ME estimates the tree-width k of the graph, constructs its tree decomposition or, if necessary, performs a reduction to a partial k-tree with given overhead while the BE can automatically instantiate extra synchronization nodes when needed. All this code & data is used to steer the *on-line* resource managers in the OS in making better run-time decisions.

4 Applications and implementation

This section details the implementation of the LIME tool-chain as well as to two real-life application use-cases, one from the embedded modem processing RT domain - SDR, and another one from HPC GPGPU computing domain - Compute Unified Device Architecture (CUDA).

4.1 Tool implementation

In the prototype LIME flow, we have chosen a *pragmatic* approach to building software. Rather than e.g., implementing our own parser for C99, we:

- re-used GCC [20] C *compiler's* `-fdump-translation-unit` option. Since no regular plug-in interface is currently standard in GCC, we chose to rely on this compiler debugging option to retrieve the parse trees.
- implemented the *FE* engine as an AWK script that extracts signatures and data-types from the dump generated by the GCC and saves them as XML.
- used ad-hoc XML *parsing framework* to process user-supplied GXF graphs as well as generated intermediate XML files.
- used an existing SDF analysis *ME* engine implemented in OCAML [35,34].
- implemented *BE* as another AWK script that generates SDR or CUDA specific C code from intermediate XML.

This sequence is initiated from `slimer`, which is built as a shell script that encapsulates a collection of Makefiles and other scripts. We plan to rewrite these *ad-hoc* scripting solutions using a single programming/scripting environment.

4.2 SDR use-case

Next generation 4G as well as current 3G wireless standards force system suppliers to start looking into *programmable radio computers*, where the hardware deals with Radio Frequency (RF) *front-end processing* and provides *raw compute capability* that can be utilized to run one, or more software baseband modems simultaneously. This requires an extensive software support on the level of the *infrastructure*, where modem processing tasks can be started and stopped, can communicate, synchronize, and can be *composed* into radio applications with *predictable* properties.

One such baseband platform is currently being developed at NXP, utilizing a heterogeneous MPSoC comprising a number of *ARM cores* and a number of *Embedded Vector Processor (EVP) cores*, having both *shared-memory* and *message-passing* primitives (via dedicated *DMA units*) to assist data-transfers to and from the host Application Processor (AP), which is handling higher-level stacks such as the Internet Protocol and User Interface (UI).

The Sea-of-DSP (SoD) software that runs on top of this platform contains a *lightweight streaming kernel* that implements task scheduling, FIFO communication, and synchronization primitives (not too different from POSIX ones depicted in Fig. 1), as well as a *Network Manager (NM)* that is used to start/stop tasks, configure them, and to setup the FIFO channels. The tasks & radio applications are programmed in C, typically directly using proprietary streaming kernel & NM APIs, which can be difficult to learn, use, maintain, and port.

```
extern void PROCESS(const int buf[5],int obuf[5]);
int COPY_SHELL (void) {
  int buf[5],obuf[5];
  if (!SELECTIN(0, ((5)*size(int))))
    return BLOCKED;
  if (!SELECTOUT(0, ((5)*size(int))))
    return BLOCKED;
  DPRINTF("reading_port_%i\n",0);
  READ(0, buf, ((5)*size(int)));
  PROCESS((const int*)buf, obuf);
  DPRINTF("write_port_%i\n",0);
  WRITE(0, obuf, ((5)*size(int)));
  return OK;
}
```

Fig. 11. Generated SoD shell with *non-blocking* primitives and *double-buffering*.

LIME offers an attractive alternative to proprietary APIs because baseband modem *suppliers* prefer to focus on their core business and allow *integrators* to map modems to specific MPSoCs. This requires a fair degree of *platform-independence* as well as *binary component* delivery. Also, modems require analytical properties that guarantee deadlock freedom and minimized resource usage.

As our initial prototype shows, all of these are guaranteed with LIME, where the BE is able to generate *shell wrapper* code dealing with kernel primitives as well as code/data related to the NM setup, see for example Fig. 11 which shows generated shell for a copy component from Fig. 5. This proves that the modems are flexibly yet efficiently *isolated* from details of a particular platform.

4.3 CUDA use-case

CUDA is a system architecture from NVidia, which is now emerging as a new player in the HPC domain. It builds on a vision of a massive multi-core platform (latest offerings comprising 240 cores per chip), organized in *clusters* of 8 *SIMT cores* in MIMD mode. Each core has a *register file*, and each cluster has *local shared-memory* on-chip. The GPU has access to *global shared-memory* on-board as well as cached access to *constant-* and *texture-memories*.

The programming environment offered by CUDA is layered: the C-based *kernel* programming language provides low-level *atomics* and *inter-thread* synchronization. The C++-based *runtime* API provides high-level *host* interface to issue memory transfers to/from the Graphics Processing Unit (GPU), launch computational *kernels*, and synchronize. The lower-level C-based *driver* API serves similar purposes, but allows a more verbose but direct control of the GPU.

```
#include CUDA
__device__ void PROCESS(buf,obuf)
  const int buf[restrict 5];
  int obuf[restrict 5];
{
  for(int i=0; i< 10; i++)
    obuf[i]=buf[i];
}
```

```
#define TID\
  (blockIdx.x*blockDim.x+threadIdx.x)
__global__ void COPY_KERNEL(buf,obuf)
  const unsigned int buf[restrict];
  unsigned int obuf[restrict];
{
  PROCESS(&buf[TID*5],&obuf[TID*5]);
}
```

Fig. 12. The copy *lime* and the generated CUDA *kernel* (host code not shown).

LIME fits well with the CUDA software infrastructure, especially its lower-level driver API, because all communication, kernel startup, and synchronization primitives are *implicit* in the *lime* code and are filled-in at compile time by the CUDA BE. In fact, *lime* is conceptually identical to CUDA's __device__ *function* - both are implemented as C functions and produce/consume data via function arguments. The complexity of *data distribution* and *inter-thread synchronization* is hidden by LIME in the component shell. Such a shell is mapped to a CUDA *computational kernel* (i.e., __global__ function callable from the host), which is responsible of mapping chunks of data to an appropriate __device__ function and call-out to that function, see Fig. 12.

Because LIME is capable of generating both the *host* code as well as the *device* code, the complexity of application development with LIME is greatly reduced in comparison to direct usage of the CUDA *runtime* API. This is especially applicable for *streams* and/or multiple GPU *contexts*, as CUDA requires context of each GPU to run on a different OS thread [36]. In the LIME CUDA BE, we directly use driver API calls, allowing more aggressive optimizations than what is possible with the CUDA runtime API.

5 Conclusions & Future work

The freedom to map LIME algorithms to either shared-, distributed-memory or message-passing architectures lies in the fact that despite that each algorithm's dependencies are explicit like in MPI, they are still expressed as pointers like in OMP (see Subsection 3.2). The freedom to optimize LIME graphs specifically for each computing domain lies in the fact that no API calls are mandated and

that the tool-chain is not constrained by semantics associated to such calls and has the freedom to generate virtually any shell. Even the overhead of a call from the *shell wrapper* code to a group of *lime*s implementing an algorithm can be optimized away automatically by forcing inlining in the C compiler.

The input to the LIME flow addresses only the functional aspect of the component and hence it is possible to document it well (e.g., using *literate programming* [28]), and to encourage designers to keep the documentation in sync with the implementation in the same LIME source, using C and GXF.

LIME has been shown to allow effective generation of 60-70% of the C code from a LIME source, for typical embedded RT codes. The generated wrapper code does not introduce any additional overhead - it is equivalent to hand-written code both functionally and in performance. Wrappers only contain calls to the underlying OS kernel, calls to the compiled representation of the algorithms, and some debugging aids (see Figures 11 and 12). Overall, the code contains therefore only 30-40% that implements the core algorithms (isolated as *lime*s manually), with the 60-70% being platform-specific glue (generated as shells).

Decoupling the generated wrapper source code from the binary representation of the functional block at the linker level allows 3rd party delivery of binary components, which is an essential mechanism for Intellectual Property (IP) protection and suits current industrial development practices well, while still supporting hardware-dependent parallelization to take place after IP delivery.

LIME supports an evolutionary approach to building parallel systems in the embedded RT and HPC domains by leveraging an existing programming language, C, which is well-known and widely used in the community. Although no intrusive modifications are proposed by LIME, it encourages designers to split their algorithms into variable-grained algorithmic components, *lime*s, which are still expressed using standard C. This increases *scalability* as well as *analysability* and provides better opportunities for platform-specific optimization. We believe that this is a promising way towards a model of parallel computing that is leveraging on legacy technology at the same time as being future-proof.

Future work includes a LIME BE for pthreads, direct comparison with MPI and OMP, implementation of *cooperative scheduling*, *dynamic allocation*, Multi-Dimensional DF (MDDF), relaxed *type matching* as well as improved ME analysis techniques for partial k-trees. Also, we have planned to work on *visual graph editors*, and support for other sequential languages such as C++ and C#.

Thanks: whole SDR team (in particular: D. van Kampen, M. van Splunter and K. van Berkel) for providing challenging use-cases, SW Infra team (in particular: Jack Goossen and Clara Otero Pérez) for providing valuable feed-back.

References

1. Charm. `http://charm.cs.uiuc.edu/research/charm/`.
2. Esterel. `http://www.esterel-technologies.com/technology/WhitePapers/`.

3. Graph exchange language. http://www.gupro.de/GXL/.
4. Lustre. http://www-verimag.imag.fr/SYNCHRONE/.
5. Message passing interface. http://www.mpi-forum.org.
6. Openmp. http://www.openmp.org.
7. Parallel virtual machine. http://www.csm.ornl.gov/pvm/pvm_home.html.
8. Streamit. http://www.cag.lcs.mit.edu/streamit/.
9. S. Arnborg, D. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Matrix Analysis and Applications*, 8(2):277–284, 1987.
10. K. Asanovic et al. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department University of California, Berkeley, Dec. 2006.
11. AT&T. Graphviz. http://graphviz.org/.
12. P. Bellens et al. Cellss: A programming model for the cell be architecture. In *Proceedings ACM/IEEE SC 2006 Conference*, Nov. 2006.
13. K. v. Berkel et al. Vector processing as an enabler for software-defined radio in handheld devices. *EURASIP Journal on Applied Signal Processing*, (16), 2005.
14. Berkeley. Ptolemy. http://ptolemy.eecs.berkeley.edu/.
15. G. Blisen et al. Cyclo-static dataflow. In *IEEE Transactions on Signal Processing*, volume 44, pages 397–408, 1996.
16. H. L. Bodlaender. Dynamic programming on graphs of bounded treewidth. 317:105–118, 1988. Lecture Notes in Computer Science.
17. J. Buck. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*. PhD thesis, Univ. of California, Berkeley, September 1993.
18. Cilk. Cilk. http://www.cilk.com/.
19. CodePlay. Sieve. http://www.codeplay.com/technology/sieve.html.
20. FSF. Gnu compiler collection. http://gcc.gnu.org/.
21. A. Ghamarian et al. Throughput analysis of synchronous data flow graphs. In *ACSD*, pages 25–34, June 2006.
22. A. Ghuloum et al. Future-proof data parallel algorithms and software on intel multi-core architecture, Nov. 2007.
23. A. Gonzalez-Escribano. *Synchronization Architecture in Parallel Programming Models*. PhD thesis, Dept. Informatica, 2003. http://www.infor.uva.es/~arturo/home.eng.html.
24. R. Govindarajan et al. Minimizing memory requirements in rate-optimal schedules. In *ASAPS*, pages 75–86, Aug. 1993.
25. S. Ha and E. Lee. Compile-time scheduling of dynamic constructs in dataflow program graphs. *IEEE Transactions on Computers*, 46(7):768–778, July 1997.
26. HPCL. Unified parallel c. http://upc.gwu.edu/.
27. M. Josep et al. A flexible and portable programming model for smp and multi-cores. Technical report, Barcelona Supercomuputing Center, June 2007.
28. D. Knuth. *Literate Programming*. Number 27. CSLI, 2nd edition, 1992.
29. E. Kock et al. YAPI: Application modeling for signal processing systems. In *Proc. Design Automation Conference (DAC)*, pages 402–405, Los Angeles, June 2000.
30. A. Kukanov et al. The foundations for scalable multi-core software in intel threading building blocks, Nov. 2007.
31. R. Kumar, D. Tullsen, N. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, Nov. 2005.
32. E. Lee and D. Messerschmitt. Sdf. In *Proceedings of the IEEE*, 1987.
33. E. Lee and D. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. In *IEEE Transactions on Computers*, 1987.

34. O. Moreira and M. Bekooij. Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing*, 2007.
35. O. Moreira, F. Valente, and M. Bekooij. Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *Proc. Embedded Software Conference (EMSOFT)*, October 2007.
36. NVidia. Cuda. `http://www.nvidia.com/object/cuda_home.html`.
37. RapidMind. Sh. `http://rapidmind.com/technology.php#programming`.
38. R. Reiter. Scheduling parallel computations. *Journal of the ACM*, Oct. 1968.
39. N. Robertson and P. Seymour. Graph minors. i. excluding a forest. *Journal of Combinatorial Theory Series B*, (35):39–61, 1983.
40. A. Varbanescu et al. Sp@ce - an sp-based programming model for consumer electronics streaming applications. In *LCPC 2006*, pages 33–48. Springer, Nov. 2006.

A Acronyms

ANSI	American National Standards Institute
AP	Application Processor
API	Application Programming Interface
ARM	Acorn RISC Machine (or Advanced RISC Machine)
AWK	Aho, Weinberger, Kernighan (text processor)
BB	Base-Band
BDF	Boolean DF
BE	Back-End
CE	Consumer Electronics
CellSs	Cell Superscalar
CF	Control-Flow
CSDF	Cyclo-Static DF
CTM	Close To Metal
CUDA	Compute Unified Device Architecture
DDF	Dynamic DF
DF	Data-Flow
DLP	Data-Level Parallelism
DMA	Direct Memory Access
DSP	Digital Signal Processor
EVP	Embedded Vector Processor
FE	Front-End
FIFO	First-In First-Out
FW	Framework
GCC	GNU Compiler Collection (formerly GNU C Compiler)
GPGPU	General Purpose GPU
GPU	Graphics Processing Unit
GXF	Graph Exchange Format
GXL	Graph Exchange Language
HPC	High-Performance Computing
HPF	High-Performance Fortran
HSDF	Homogeneous Synchronous DF
HW	Hardware
ILP	Instruction-Level Parallelism
IP	Intellectual Property
LIME	Less Is More
MDDF	Multi-Dimensional DF
ME	Middle-End
MIMD	Multiple-Instruction Multiple-Data
MLP	Memory-Level Parallelism
MPI	Message Passing Interface
MPSoC	Multi-Processor SoC
NM	Network Manager
NXP	Next Experience Semiconductors
OCAML	Objective CAML
OMP	OpenMP
OS	Operating System
PL	Programming Language
PM	Programming Model
POSIX	Portable Operating System Interface
PPM	Parallel Programming Model
pthreads	POSIX threads
PVM	Parallel Virtual Machine
RF	Radio Frequency
RT	Real-Time
SAC	Single Assignment C
SDF	Synchronous DF
SDR	Software-Defined Radio
SIMT	Single-Instruction Multiple-Thread
SK	Streaming Kernel
SMPSs	SMP Superscalar
SoD	Sea-of-DSP
SP	Series-Parallel
STL	Standard Template Library
SW	Software
TBB	Threading Building Blocks
TLP	Task-Level Parallelism
UI	User Interface
UNIX	Unified Information and Computing System
VLA	Variable-Length Array
XML	Extensible Markup Language
YAPI	Y-API