The SPIRE Methodology of Compiler Parallel Intermediate Representation Design

Dounia Khaldi[‡], Pierre Jouvelot[¶], François Irigoin[¶], Corinne Ancourt[¶] and Barbara Chapman[‡]

[‡]Department of Computer Science, University of Houston
Houston, Texas 77004
{dkhaldi,bchapman}@uh.edu
[¶]MINES ParisTech, PSL Research University, France
firstname.lastname@mines-paristech.fr

Abstract

SPIRE is the first incremental methodology for designing the intermediate representations (IR) of compilers that target parallel programming languages. Its core philosophy is to extend in a systematic manner the IRs found in the compilation frameworks of sequential languages. Avoiding the often-used ad-hoc approach of encoding all parallel constructs as "fake" function calls, SPIRE enables the leveraging of current compilers for sequential languages to address both control and data parallel constructs while preserving as much as possible the correctness of existing analyses for sequential code. This upgrading process is formalized as an "intermediate representation transformer" at the syntactic and semantic levels. We show this can be done via the addition of only three global parallel traits on top of any sequential IR, namely execution, synchronization and data distribution, precisely defined via a formal semantics and rewriting rules.

We use the sequential intermediate representation of PIPS, a comprehensive source-to-source compilation platform, as a use-case for our approach. We introduce SPIRE parallel primitives, extend PIPS intermediate representation and show how example code snippets from different programming languages can be represented this way. A formal definition of SPIRE operational semantics is provided, built on top of the one used for the sequential intermediate representation.

We assess the generality of our proposal by showing how different sequential IRs, namely the LLVM and WHIRL IRs of the widely used LLVM and Open64-based OpenUH compilers, can be systematically extended to handle parallelism using the SPIRE methodology. Experimentally, SPIRE has been implemented in PIPS, and the resulting parallel IR used successfully to perform OpenMP and MPI parallel code generation as part of the PIPS parallelization process; it is also being implemented within the LLVM compiler to validate its use in parsing explicitly-parallel OpenSHMEM programs and enabling important optimizations for one-sided communications using loop tiling and communication vectorization.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers

Keywords Parallel language intermediate representation, Compilers, Operational semantics, PIPS, LLVM, OpenUH, Open64

1. Introduction

The growing importance of parallel computers and the search for an efficient programming model led, and is still leading, to a proliferation of parallel programming languages such as, currently, Cilk [37], Chapel [7], X10 [5], Habanero-Java [9], OpenMP [6], CAF [33], OpenSHMEM [4] or MPI [3]. To adapt to such an evolution, compilers need to be equipped with internal intermediate representations (IR) for parallel programs. The choice of a proper parallel IR (PIR) is of key importance, since the efficiency and power of the transformations and optimizations these compilers can perform are closely related to the selection of a proper program representation paradigm.

How are current compilers surviving without PIR? A common technique for introducing parallel constructs in sequential IRs, adopted for instance by GCC, LLVM [27] and Open64-based OpenUH [28], is to encode them as "fake" function calls; although simple, this ad-hoc solution prevents high-level reasoning about parallel languages while possibly introducing semantic inconsistencies in existing sequential program analyses. Worse, users implicitly rely upon the lack of interprocedural analyses within many compilers to preserve the semantics of programs even in the presence of code optimizations. This approach clearly is not satisfactory and, given the wide variety of existing programming models, it would be better, from a software engineering point of view, to find a parallel-specific IR, hopefully as general and simple as possible.

Existing proposals for program representation techniques already provide a basis for the exploitation of parallelism via the encoding of control and/or data flow information. HPIR [38] or InsPIRe [20] are instances that operate at a high abstraction level, while the hierarchical task, stream or program dependence graphs (we survey these notions in Section 6) are better suited to graph-based approaches. Yet many more existing compiler frameworks use traditional representations for sequential-only programs [36], and changing their internal data structures or adding specific built-ins to deal with parallel constructs is a difficult and time-consuming task. The general methodology introduced in this paper, which extends sequential intermediate representations into as structured as possible parallel ones, strives to minimize the number of introduced built-ins, which hinder program analyses, while allowing the expression of as much parallelism as possible.

Why do we need Incremental Parallel Intermediate Representations? The main practical motivation of our proposal is to preserve the many years of development efforts invested in huge compiler platforms such as GCC (more than 7 million SLOC), LLVM (more than 1 million SLOC), PIPS [19] (600,000 SLOC), OpenUH (more than 3 million SLOC)... when upgrading their intermediate representations to handle parallel languages. We provide an evolutionary path for these large software developments via the introduction of the Sequential to Parallel Intermediate Representation Extension (SPIRE) methodology that we show that can be plugged into existing compilers in a rather simple manner.

SPIRE is based on only three key concepts for extension: (1) the parallel vs. sequential execution of groups of statements such as sequences, loops and general control-flow graphs, (2) the global synchronization characteristics of statements and the specification of finer grain synchronization via the notion of events and (3) the handling of data distribution for different memory models. To describe how this approach works, we use SPIRE to extend the intermediate representation (IR) [11] of PIPS, a comprehensive source-to-source compilation and optimization platform, and illustrate its generality by applying SPIRE to LLVM, a widespread SSA-based compilation infrastructure, and OpenUH, an Open64-based compiler for parallel languages.

The design of SPIRE is the result of many trade-offs between generality and precision, abstraction and low-level concerns. On the one hand, and in particular when looking at source-to-source optimizing compiler platforms adapted to multiple source languages, one needs to be able to represent as many of the existing (and, hopefully, future) parallel constructs while minimizing the number of new concepts introduced in the parallel IR. Yet, keeping only a limited number of hardware-level notions in the IR, while good enough to deal with all parallel constructs, could entail convoluted rewritings of some high-level parallel flows. We used an extensive survey of key parallel languages, namely Cilk, Chapel, X10, Habanero-Java, OpenMP and CAF and libraries, namely MPI and OpenSHMEM, to guide our design of SPIRE, while showing how to express their relevant parallel constructs within SPIRE.

The four main contributions of this paper are:

- SPIRE, a new, simple, parallel intermediate representation extension methodology for designing the parallel IRs used in compilation frameworks;
- the small-step, operational semantics of the SPIRE transformation process, to formally define how its key parallel concepts are added to existing systems;
- an evaluation of the generality of SPIRE, by showing how the SPIRE methodology can be applied to the IRs of PIPS, LLVM and OpenUH.
- the experimental application of SPIRE in PIPS, yielding a parallel IR that has been successfully used for both automatic OpenMP and MPI task-level parallelization; a second implementation, into the LLVM compiler, is being performed to validate SPIRE for the parsing of the parallel language OpenSHMEM and enabling important optimizations for one-sided communications using loop tiling and communication vectorization.

After this introduction, we describe our use-case sequential IR, part of the PIPS compilation framework, in Section 2. Our parallel extension proposal, SPIRE, is introduced in Section 3, where we also show how simple illustrative examples written in Habanero-Java, CAF, Chapel and OpenSHMEM can be easily represented within SPIRE. The formal operational semantics of SPIRE is given in Section 4. Section 5 illustrates the generality of SPIRE by applying it on LLVM and WHIRL and discusses performance results. We survey existing parallel IRs in Section 6. We discuss future work and conclude in Section 7.

2. PIPS (Sequential) IR

Since this paper introduces SPIRE as an *extension* formalism for existing intermediate representations, a sequential, base case IR is needed to present our proposal. We chose the IR of PIPS, a source-to-source compilation and optimization platform [19], to showcase our approach, since it is readily available, well-documented and encodes both control and data dependences. To help support our claim of the generality of our approach, Section 5.1 illustrates two other applications of SPIRE, on LLVM [27] and WHIRL [2].

We provide in Figure 1 a high-level description of a slightly simplified subset of the intermediate representation of PIPS, the part that is directly related to the parallel paradigms in SPIRE. It is specified using Newgen [22], a Domain Specific Language for the definition of set equations from which a dedicated API is automatically generated to manipulate (creation, access, IO operations...) data structures implementing these set elements.

Control flow in PIPS IR is represented via instructions, members of the disjoint union (+) set `instruction`. An instruction can be either a simple call or a compound instruction, i.e., a `for` loop, a sequence or a control flow graph. A call instruction represents built-in or user-defined function calls; for instance, assign statements are represented as calls to the "`:=`" function. The `call` set is not defined here.

Instructions are included within statements, which are members of a cartesian product set (`x`) that also incorporates the declarations of local variables; thus a whole function is represented in PIPS IR as a statement. In Newgen, a given set component can be distinguished using a prefix such as `declarations` here; all named objects such as user variables or built-in functions in PIPS are members of the `entity` set (the `value` set denotes constants while the "`*`" symbol introduces Newgen list sets).

Compound instructions can be either (1) a loop instruction, which includes an iteration index variable with its lower, upper and increment expressions and a loop body (the `expression` set definition is not provided here), (2) a sequence of statements, encoded as a list, or (3) a control, for control flow graphs.

Programs that contain structured (`continue`, `break` and `return`) and unstructured (`goto`) transfers of control are handled in the PIPS intermediate representation via the `control` set. A `control` instruction has one entry and one exit node; a `node` in a graph is labeled with a statement and its lists of predecessor and successor control nodes. Executing a `control` instruction amounts to following the control flow induced by the graph successor relationship, starting at the entry node, while executing the node statements, until the exit node is reached.

```
instruction = call + forloop + sequence + control;
statement   = instruction x declarations:entity*;
entity      = name:string x type x storage;
storage     = return:entity + ram + rom:unit;
forloop     = index:entity x lower:expression x
              upper:expression x step:expression x
              body:statement;
sequence    = statements:statement*;
control     = entry:node x exit:node;
node        = statement x predecessors:node* x
              successors:node*;
```

Figure 1: Simplified Newgen definitions of the PIPS IR

3. SPIRE, a Sequential to Parallel IR Extension Methodology

In this section, we present in detail the SPIRE methodology, which can be used to add parallel concepts to sequential IRs. After introducing our design philosophy, we describe the application of SPIRE on the PIPS IR. We illustrate these SPIRE-derived constructs with code excerpts from various parallel programming languages; our intent is not to provide here general rewriting techniques from these to SPIRE (this would be way out of the scope of this paper), but to provide hints on how such rewritings might possibly proceed. In Section 5.1, using LLVM and WHIRL, we show that our approach is general enough to be adapted to other IRs.

3.1 Design Approach

SPIRE intends to be a practical methodology to extend existing sequential IRs to adapt to parallelism issues, either to generate parallel code from sequential programs or address explicitly parallel programming languages. Interestingly, the idea of seeing the issue of parallelism as an extension over sequential concepts is in sync with Dijkstra's view that "parallelism or concurrency are operational concepts that refer not to the program, but to its execution." [14]. If one accepts such a vision, adding parallelism extensions to existing IRs, as advocated by our approach with SPIRE, can thus, at a fundamental level, not be seen as an afterthought but as a consequence of the fundamental nature of parallelism.

Our design of SPIRE does not pretend to be minimalist but to be as seamlessly as possible integrable within actual IRs, while able to handle as many parallel programming constructs as possible. To be successful, our design point must provide proper trade-offs between generality, expressibility and conciseness of representation. We used an extensive survey of existing parallel languages to guide us during this design process. Table 1, which extends the one provided in [25], summarizes the main characteristics of eight recent and widely used parallel languages and libraries: Chapel, X10, Habanero-Java, OpenMP, Cilk, CAF, OpenSHMEM and MPI. The main constructs used in each language to launch task and data parallel computations, perform synchronization, introduce atomic sections and transfer data in the various memory models are listed. Our main finding from this analysis is that, to be able to deal with parallel programming, one needs to add to a given sequential IR the ability to specify (1) the parallel execution of groups of statements, (2) the synchronization between statements and (3) data layout, i.e., how memory is modeled in a given parallel language.

The last line of Table 1 shows that SPIRE is based on the introduction of only ten key notions, collected in three groups: (1) execution, via the `parallel` and `reduced` constructs; (2) synchronization, via the `spawn`, `barrier`, `atomic` and event constructs; and (3) data distribution, via `send`, `recv` and `location` constructs.

3.2 Execution

The issue of parallel vs. sequential execution appears when dealing with groups of statements, which in our case study correspond to members of the `forloop`, `sequence` and `control` sets. To apply SPIRE to PIPS sequential IR, an `execution` attribute is added to these sequential set definitions:

```
forloop'       = forloop x execution;
sequence'      = sequence x execution;
control'       = control x execution;
```

The primed sets `forloop'` (expressing data parallelism) and `sequence'` and `control'` (implementing control parallelism) represent SPIREd-up sets for the PIPS parallel IR. Of course, the 'prime' notation is used here for pedagogical purpose only; in practice, an execution field is added in the existing IR representation. The definition of `execution` is straightforward:

```
execution = sequential:unit +
            parallel:scheduling + reduced:unit;
scheduling = static:unit + dynamic:unit +
             speculative:unit + default:unit
```

where `unit` denotes a set with one single element; this encodes a simple enumeration of cases for execution. A `parallel` (resp. `reduced`) execution attribute asks for all loop iterations, sequence statements and control nodes of `control` statements to be all launched in parallel, in an implicit flat fork/join (resp. left-to-right, tree-like) fashion. SPIRE is an extendable framework; this can be shown with the generalization of the definition of `parallel` domain in order to handle other features such as dynamic scheduling, speculation, etc.

An example, in the left side of Figure 2, from Chapel, illustrates its `forall` data parallelism construct and a `reduce` operation, which can be encoded with a SPIRE parallel loop and a SPIRE reduced loop.

```
forall i in 1..n do
  t[i] = i;
var (sumVal) = + reduce
  ([i in 1..n] f(i),
   1..n);
```

```
forloop(i,1,n,1,
        t[i] = i,
        parallel(default));
forloop(i,1,n,1,
  sumVal = sumVal+f(i),
  reduced)
```

Figure 2: SPIRE version of `forall` and `reduce` in Chapel

3.3 Synchronization

The issue of synchronization is a characteristic feature of the runtime behavior of one statement with respect to other statements. In parallel code, one usually distinguishes between two types of synchronization: (1) collective synchronization between threads using barriers, and (2) point-to-point synchronization between participating threads. We suggest this can be done in two parts.

3.3.1 Collective Synchronization

SPIRE extends sequential intermediate representations in a straightforward way by adding a synchronization attribute to the specification of statements:

```
statement' = statement x synchronization;
```

Coordination by synchronization in parallel programs is often managed via coding patterns such as barriers, used for instance when a code fragment contains many phases of parallel execution where each phase should wait for the precedent ones to proceed. We define the `synchronization` set via high-level coordination characteristics useful for optimization purposes:

```
synchronization = none:unit + spawn:entity +
                  barrier:unit + atomic:reference;
```

Assume S is the statement with a synchronization attribute:

- `none` specifies the default behavior, i.e., independent with respect to other statements, for S;
- `spawn` induces the creation of an asynchronous task S, while the value of the corresponding entity is the user-chosen number of the thread that executes S. By convention, we say that `spawn` creates processes in the case of the message-passing and PGAS memory models, and threads, in case of the shared memory model;

Language/ Library	Execution		Synchronization				Memory	
	Parallelism	Reduction	Task creation	Task join	Atomic section	Point-to-point	Model	Data distribution
Chapel (Cray)	forall coforall cobegin	reduce	begin on	sync	sync atomic	sync	*PGAS (Locales)*	*implicit*
X10 (IBM)	foreach	reduce	async at future	finish	atomic	next force	*PGAS (Places)*	*implicit*
Habanero-Java (Rice)	foreach	*accumulator*	async at future	finish	atomic isolated	next get	*PGAS (Places)*	*implicit*
OpenMP	omp for omp sections	omp reduction	omp task omp section	omp taskwait omp barrier	omp critical omp atomic	—	*Shared*	private, shared...
Cilk (MIT)	—	—	spawn	sync	cilk_lock	—	*Shared*	—
CAF (Cray)	*implicit*	co_reduce	—	sync all sync images	critical end critical	event post event wait	*PGAS (Images)*	[]
OpenSHMEM (UH & ORNL)	start_pes	*shmem_sum shmem_max ...*	—	shmem_barrier_all shmem_barrier	shmem_set_lock shmem_clear_lock	shmem_wait shmem_wait_until	*PGAS (PEs)*	shmem_put shmem_get
MPI	MPI_Init	MPI_Reduce	MPI_Spawn	MPI_Finalize MPI_Barrier	—	—	*Distributed*	MPI_Send MPI_Recv...
SPIRE	parallel	reduced	spawn	barrier	atomic	signal wait	*Shared, Distributed, PGAS*	send, recv, location

Table 1: Mapping of SPIRE to parallel languages constructs (*implicit* means the compiler will insert launch code)

- `barrier` specifies that all the child threads spawned by the execution of S are suspended before exiting until they are all finished;
- `atomic` predicates the execution of S to the acquisition of a lock to ensure exclusive access; at any given time, S can be executed by only one thread. Locks are logical memory addresses, represented here by a member of the PIPS IR `reference` set (not specified in this paper).

3.3.2 Event API: Point-to-Point Synchronization

Handling point-to-point synchronization using decorations on abstract syntax trees is too constraining when one has to deal with a varying set of threads that may belong to different parallel parent nodes. Thus, SPIRE deals with this last class of coordination by introducing new values, of type `event`. SPIRE extends the underlying type system of the existing sequential IRs with a new basic type, namely `event`:

```
type' = type + event:unit ;
```

Values of type `event` are counters, in a manner reminiscent of semaphores [13]. The programming interface for events is defined by the following functions:

- `event newEvent(int i)` is the creation function of events, initialized with the integer `i` that specifies how many threads can execute `wait` on this event without being blocked;
- `void signal(event e)` increments by one the event value of `e`;
- `void wait(event e)` blocks the thread that calls it until the value of `e` is strictly greater than 0. When the thread is released, this value is decremented by one.

Note that the `void` return type will be replaced by `int` in practice, to enable the handling of error values, and that a free function may be needed in some languages.

In a first example of possible use of this event API, the construct `future` used in X10 can be seen as the spawning of the computation of `foo()`. The end result is obtained via the call to the `force` method; such a mechanism can be easily implemented in SPIRE using an event attached to the running task; it is `signal`ed when the task is completed and `wait`ed by the `force` method.

A second example, taken from Habanero-Java, illustrates how point-to-point synchronization primitives such as phasers and the `next` statement can be dealt with using the Event API (see Figure 3, left). The `async phased` keyword can be replaced by `spawn`. In this example, the `next` statement is equivalent to the following sequence:

```
signal(ph); wait(ph); signal(ph);
```

where the event `ph` is initialized to `newEvent (-(n-1))`. The second `signal` is used to resume the suspended tasks in a chain-like fashion.

```
finish{
 phaser ph = new phaser();
 for(j = 1;j <= n;j++){
  async phased(
          ph<SIG_WAIT>){
   S;
   next;
   S';
  }
 }
}
```

```
barrier(
 ph = newEvent(-(n-1));
 j = 1;
 loop(j <= n,
  spawn(j,
        S;
        signal(ph);
        wait(ph);
        signal(ph);
        S');
  j = j+1))
```

Figure 3: SPIRE variant of a phaser in Habanero-Java

Even though our proposal based on events is able to represent high-level point-to-point synchronization constructs such as phasers, its admittedly low level of abstraction makes dealing with source-to-source optimization algorithms difficult. The phaser example illustrates the kind of trade-offs we had to make when adding as much parallelism as possible to sequential IRs without introducing too many specific parallel-related constructs via SPIRE.

3.4 Data Distribution

The ability of handling the various memory models used by parallel languages is an important issue when designing a generic intermediate representation. One currently considers there are three

main parallel memory models: shared, message passing and, more recently, PGAS. The Partitioned Global Address Space memory model, which appears in languages such as CAF, Habanero-Java, X10 and Chapel, introduces various new notions such as `image`, `place` or `locale` to label portions of a logically-shared memory that processes may access, in addition to complex APIs for distributing data over these portions.

3.4.1 Memory Information

SPIRE is able to handle all three memory models using specific memory information, namely private, shared and *pgas*. Each process has its own private memory; the address of a shared variable refers, within each thread, to the same physical memory location; pgas memory is distributed evenly among different processes. In this last case, e.g., in OpenSHMEM and CAF, all processes have their own view of pgas memory.

To handle memory information, SPIRE extends the definition of the storage feature of entities by specifying where a given entity is stored, i.e., within private, shared or pgas memory. This is done by adding a `location` domain to the `storage` domain of PIPS IR:

```
storage' = storage x location;
location = private:unit + shared:unit +
           pgas:unit;
```

Since SPIRE is designed to extend existing IRs for sequential languages, it can be straightforwardly seen as using, by default, a shared memory model when parallel constructs are added.

We show below two examples related to pgas memory. First, in CAF, coarrays, which are extension of Fortran arrays, are pgas arrays. A coarray has codimensions, which specify the image to which it belongs. In the following example, the `dest` coarray of 20 elements will be visible and accessible remotely by all images.

```
integer(len=20) :: dest[*]
```

Our second example shows an OpenSHMEM statement allocating `dest`, which is also remotely accessible by all PEs (Processing Elements).

```
dest = (int*)shmalloc(sizeof(int)*20);
```

3.4.2 Two-Sided Memory Access

In order to take into account the explicit distribution required by the message passing memory model used in parallel languages such as MPI, SPIRE introduces the `send` and `recv` blocking functions for implementing communication between processes:

- `void send(int dest, entity buf)` transfers the value of Entity `buf` to the process numbered `dest`;
- `void recv(int source, entity buf)` receives in `buf` the value sent by Process `source`.

Non-blocking communications can be easily implemented in SPIRE using the above primitives within `spawn`ed statements. Also, broadcast collective communications, such as defined in MPI, can be seen as wrappers around `send` and `recv` operations. When the master process and receiver processes want to perform a broadcast function, then, if this process is the master, its broadcast operation is equivalent to a loop over receivers, with a call to `send` as body; otherwise (receiver), the broadcast is a `recv` function. Note that after optimization phases of the compiler, we generate back, at the backend phase, runtime library calls in order to take advantage of support for optimized lower level communication functions.

3.4.3 One-Sided Memory Access

In one-sided communications, only the source or destination process participates in asynchronous memory accesses, decoupling thus data transfer and synchronization. In order to take into account the explicit distribution required by the PGAS memory model used in parallel languages such as CAF or libraries such as OpenSHMEM, SPIRE extends the traditional semantics of memory accesses and assignments. Since the information needed for specifying remote accesses is already present in the `location` domain of entities, there is no need to gather again this information (see Section 4.2). Put operations copy data from a local source memory area to a memory area of the remote target (get are dual operations). The following function call in OpenSHMEM:

```
shmem_int_put(dest, src, 20, pe);
```

can be represented in SPIRE by:

```
for(i=1,20)
  dest{pe}[i]= src[i]
```

where `location` of `dest` is `pgas`. The entity domain is extended to handle expressions in the left hand side of assignments. Note that at code generation phase (after optimizations), this loop of assignments will ultimately be transformed back to the runtime library call `shmem_int_put(dest, src, 20, pe)`.

4. SPIRE Operational Semantics

The purpose of the formal definition described in this section is to provide a solid foundation for program analyses and transformations. It is a systematic way to specify our IR extension mechanism, something seldom present in IR definitions. It also illustrates how SPIRE leverages the syntactic and semantic level of sequential constructs to parallel ones, preserving the sequential traits and, thus, related analyses.

Fundamentally, at the syntactic and semantic levels, SPIRE is a methodology for expressing representation transformers, mapping the definition of a sequential language IR to a parallel version. We define the operational semantics of SPIRE in a two-step fashion: we introduce (1) a minimal core parallel language that we use to model fundamental SPIRE concepts and for which we provide a small-step operational semantics and (2) rewriting rules that translate the more complex constructs of SPIRE in this core language.

4.1 Sequential Core Language

Illustrating the transformations induced by SPIRE requires the definition of a sequential IR basis, as was done above, via PIPS IR. Since we focus here on the fundamentals, we use as core language a typical, minimal sequential language of statements `S` in `Stmt`, based on identifiers `I` in `Ide` and expressions `E` in `Exp`. Sequential statements are: (1) `nop` for no operation, (2) `I=E` for an assignment of `E` to `I`, (3) $\texttt{S}_1\texttt{;}\texttt{S}_2$ for a sequence and (4) `loop(E,S)` for a while loop.

At the semantic level, a statement in `Stmt` is a very simple memory transformer. A memory $m \in Memory$ is a mapping in $\texttt{Ide} \rightarrow Value$, where values $v \in Value = N + Bool$ can either be integers $n \in N$ or booleans $b \in Bool$. The sequential operational semantics for `Stmt`, expressed as transition rules over configurations $\kappa \in Configuration = Memory \times \texttt{Stmt}$, is given in Figure 4; we assume that the program is syntax- and type-correct. A transition $(m, \texttt{S}) \rightarrow (m', \texttt{S'})$ means that executing the statement S in a memory m yields a new memory m' and a new statement `S'`; we posit that the "$\rightarrow$" relation is transitive. Evaluation rules (see the case for assignments in Figure 4) encode typical small-step operational semantics for the sequential part of the core language. We assume that $\xi \in \texttt{Exp} \rightarrow Memory \rightarrow Value$ is the usual function for expression evaluation.

4.2 SPIRE as a Language Transformer

Syntax At the syntactic level, SPIRE specifies how a grammar for a sequential language such as `Stmt` is transformed, i.e., ex-

$$\frac{v = \xi(\mathtt{E})m}{(m, \mathtt{I} = \mathtt{E}) \rightarrow (m[\mathtt{I} \rightarrow v], \mathtt{nop})} \quad (1)$$

Figure 4: `Stmt` sequential transition rules (excerpt)

tended, with synchronized parallel statements. The grammar of `SPIRE(Stmt)` in Figure 5 adds to the sequential statements of `Stmt` (from now on, synchronized using the default `none`) new parallel statements: a task creation `spawn`, a termination `barrier` and two `wait` and `signal` operations on events or `send` and `recv` operations for communication. Synchronization `atomic` is defined via rewriting (see below). The statement `barrier_wait(`n`)`, added here for specifying the multiple-step behavior of the `barrier` statement in the semantics, is not accessible to the programmer. Figure 3 provides the SPIRE representation of a program example.

Note that the grammar of `Ide` is also extended to `SPIRE(Ide)` in Figure 5 to add the possibility of performing a memory access to the pgas identifier `I` located on Process `E`, noted `I{E}` . This extension of `Ide` naturally carries over to `Exp`.

```
S ∈ SPIRE(Stmt)::=
    nop | I=E | S1;S2 | loop(E,S) |
    spawn(I,S) |
    barrier(S) | barrier_wait(n) |
    wait(I) | signal(I) |
    send(I,I') | recv(I,I')
I ∈ SPIRE(Ide)::= I | I{E}
```

Figure 5: `SPIRE(Stmt)` and `SPIRE(Ide)` syntaxes

Semantic Domains As SPIRE extends grammars, it also extends semantics. The set of values manipulated by `SPIRE(Stmt)` statements extends the sequential $Value$ domain with events $e \in \mathit{Event} = N$, that encode events current values; we posit that $\xi(\mathtt{newEvent(E)})m = \xi(\mathtt{E})m$.

Parallelism is managed in SPIRE via processes (or threads). We introduce control state functions $\pi \in State = Proc \rightarrow Configuration \times Procs$ to keep track of the whole computation, mapping each process $i \in Proc = N$ to its current configuration (i.e., the statement it executes and its own view of memory) and the set $c \in Procs = \mathcal{P}(Proc)$ of the process children it has spawned during its execution.

In the following, we note $dom(\pi) = \{i \in Proc / \pi(i)$ is defined$\}$ the set of currently running processes, and $\pi[i \rightarrow (\kappa, c)]$ the state π extended at i with (κ, c). A process is said to be *finished* if and only if all its children processes, in c, are also finished, i.e., when only `nop` is left to execute: $finished(\pi, c)$ is thus equal to $\forall i \in c, \pi(i) = ((m_i, \mathtt{nop}), c_i) \wedge finished(\pi, c_i)$.

Memory Models The memory model for sequential languages is that of a unique address space for identifiers. In our parallel extension, the memory configuration for a given process or thread may vary according to whether a variable is shared or pgas, or whether message passing is used. We suggest to adopt the same semantic rules, detailed in Figure 6, to deal with all models, but introduce two additional notions to handle the required distinctions between them. First, some constructs will make more sense in one model than an other, i.e., send/receive in message passing, events in shared address spaces and pgas variables in PGAS. Second, we impose constraints on the control states π used in the operational semantics. Namely, for all threads (or processes) i and i' with $\pi(i) = ((m, \mathtt{S}), c)$ and $\pi(i') = ((m', \mathtt{S}'), c')$, we impose that, for all identifiers `I` whose storage includes a `location` that has value `shared`, one must have $m(\mathtt{I}) = m'(\mathtt{I})$. No such constraint is needed for private or pgas locations.

Semantic Rules At the semantic level, SPIRE is a transition system transformer, mapping rules such as the ones in Figure 4 to parallel, synchronized transition rules in Figure 6. A transition $(\pi[i \rightarrow ((m, \mathtt{S}), c)]) \Rightarrow (\pi'[i \rightarrow ((m', \mathtt{S'}), c')])$ means that the i-th process, when executing S in a memory m, yields a new memory m' and a new control state $\pi'[i \rightarrow ((m', \mathtt{S'}), c')]$ in which this process now will execute S'; additional children processes may have been created in c' compared to c. We posit that the "$\Rightarrow$" relation is transitive. Rule 2 is a key rule to specify SPIRE transformer behavior, providing a bridge between the sequential and the SPIRE-extended parallel semantics; all processes can nondeterministically proceed along their sequential semantics "$\rightarrow$", leading to valid evaluation steps along the parallel semantics "$\Rightarrow$". The interleaving between parallel processes in `SPIRE(Stmt)` is a consequence of (1) the non-deterministic choice of the value of i within $dom(\pi)$ when selecting the transition to perform and (2) the number of steps executed by the sequential semantics. Note that one might want to add even more non-determinism in our semantics. Indeed, Rule 1 is atomic: loading the variables in `E` and the store operation on `I` are performed in one sequential step. For lack of space, we do not provide the simple intermediate steps in the sequential evaluation semantics of Rule 1 that would have removed this artificial atomicity.

The remaining rules focus on parallel evaluation. Rules 3 and 4 extend the usual (sequential) assignment, since the accolades represent pgas identifiers. The difference with a simple assignment or access is that the operation is performed in a remote memory area. The use of the sequential expression evaluation function ξ prevents remote accesses to be embedded within expressions; this is a conscious design decision, to ensure that sequential optimizations such as strength reduction over private expressions remain valid.

In Rule 5, `spawn` adds to the state a new process n that executes `S` while inheriting the parent memory m in a fork-like manner if the process does not already exist; otherwise, n resumes its execution and then executes `S`. The set of processes spawned by n is initially equal to $\emptyset$, and n is added to the set of processes c spawned by i. Rule 6 implements a rendezvous: a new process n executes S, while Process i is suspended as long as $finished$ is not true; indeed, Rule 7 resumes execution of Process i when all the child processes spawned by n have finished.

In Rules 8 and 9, `I` is an `event`, i.e., a counting variable used to control access to a resource or to perform a point-to-point synchronization, initialized via `newEvent` to a value equal to the number of processes that will be granted access to it. Its current value n is decremented every time a `wait(I)` statement is executed and, when $\pi(\mathtt{I}) = n$ with $n > 0$, the resource can be used or the barrier can be crossed. In Rule 9, the current value n' of `I` is incremented; this is a non-blocking operation.

In Rule 10, p and p' are two processes that communicate: p sends the datum `I` to p', while this later consumes it in `I'` .

Rewriting Rules The SPIRE concepts not dealt with in the previous section are defined via their syntactic rewriting into the core language. This is the case for both the treatment of the `execution` attribute, the remaining coarse-grain synchronization constructs and non-blocking communications. For lack of space, these simple rewritings have been omitted from this paper but are detailed in [24].

5. Validation

Assessing the quality of a methodology that impacts the definition of a data structure as central for compilation frameworks as an intermediate representation is a difficult task. This section provides

$$\frac{\kappa \to \kappa'}{\pi[i \to (\kappa, c)] \Rightarrow \pi[i \to (\kappa', c)]} \tag{2}$$

$$\frac{p = \xi(\mathtt{E})m \wedge v = \xi(\mathtt{E}')m \wedge p \neq i}{\pi[i \to ((m, \mathtt{I}\{\mathtt{E}\} = \mathtt{E}'), c)][p \to ((m', \mathtt{S}'), c')] \Rightarrow \pi[i \to ((m, \mathtt{nop}), c)][p \to ((m'[\mathtt{I} \to v], \mathtt{S}'), c')]} \tag{3}$$

$$\frac{p = \xi(\mathtt{E})m \wedge v = \xi(\mathtt{I}')m' \wedge p \neq i}{\pi[i \to ((m, \mathtt{I} = \mathtt{I}'\{\mathtt{E}\}), c)][p \to ((m', \mathtt{S}'), c')] \Rightarrow \pi[i \to ((m[\mathtt{I} \to v], \mathtt{nop}), c)][p \to ((m', \mathtt{S}'), c')]} \tag{4}$$

$$\frac{n = \xi(\mathtt{I})m \wedge ((m', \mathtt{S}'), c') = (n \in \mathit{domain}(\pi))\ ?\ \pi(n) : ((m, \mathtt{nop}), \emptyset)}{\pi[i \to ((m, \mathtt{spawn(I,S)}), c)] \Rightarrow \pi[i \to ((m, \mathtt{nop}), c \cup \{n\})][n \to ((m', \mathtt{S}'\mathtt{;S}), c')]} \tag{5}$$

$$\frac{n \notin \mathit{dom}(\pi) \cup \{i\}}{\pi[i \to ((m, \mathtt{barrier(S)}), c)] \Rightarrow \pi[i \to (m, \mathtt{barrier\_wait}(n)), c)][n \to ((m, \mathtt{S}), \emptyset)]} \tag{6}$$

$$\frac{\mathit{finished}(\pi, \{n\}) \wedge \pi(n) = ((m', \mathtt{nop}), c')}{\pi[i \to ((m, \mathtt{barrier\_wait}(n)), c)] \Rightarrow \pi[i \to ((m', \mathtt{nop}), c)]} \tag{7}$$

$$\frac{n = \xi(\mathtt{I})m \wedge n > 0}{\pi[i \to ((m, \mathtt{wait(I)}), c)] \Rightarrow \pi[i \to ((m[\mathtt{I} \to n - 1], \mathtt{nop}), c)]} \tag{8}$$

$$\frac{n = \xi(\mathtt{I})m}{\pi[i \to ((m, \mathtt{signal\,(I)\,}), c)]) \Rightarrow \pi[i \to ((m[\mathtt{I} \to n + 1], \mathtt{nop}), c)]} \tag{9}$$

$$\frac{p' = \xi(\mathtt{P}')m \wedge p = \xi(\mathtt{P})m' \wedge p \neq p'}{\pi[p \to ((m, \mathtt{send(P}', \mathtt{I})), c)][p' \to ((m', \mathtt{recv(P}, \mathtt{I}')), c')] \Rightarrow \pi[p \to ((m, \mathtt{nop}), c)][p' \to ((m'[\mathtt{I}' \to m(\mathtt{I})], \mathtt{nop}), c')]} \tag{10}$$

Figure 6: `SPIRE(Stmt)` transition rules

two possible ways to perform such an assessment on SPIRE: (1) we illustrate how it can be easily applied on other IRs, namely those of LLVM and Open64-based OpenUH, by extending their respective sequential IRs with minimal changes, thus providing support regarding the generality of our methodology, and (2) we provide information regarding its impact on run-time performance data for parallelization and optimizations.

5.1 SPIRE Application to LLVM IR

LLVM [27] (Low-Level Virtual Machine) is an open-source compilation framework that uses an intermediate representation in Static Single Assignment (SSA) [12] form. Polly [18] is a high-level loop and data-locality optimizer for LLVM. We chose the IR of LLVM to illustrate a second time our approach since LLVM has been widely used in both academia and industry. Another interesting feature of LLVM IR, compared to PIPS, is that it sports a graph approach, while PIPS is abstract syntax tree-based; each function is structured in LLVM as a control-flow graph (CFG). Figure 7 provides the definition of a significant subset of the sequential LLVM IR described in [29], written in Newgen to keep notations simple in this paper:

- a function is a list of basic blocks, which are portions of code with one entry and one exit points;
- a basic block has an entry label, a list of ϕ nodes and a list of instructions, and ends with a terminator instruction;
- ϕ nodes, which are the key elements of SSA, are used to merge the values coming from multiple basic blocks. A ϕ node is an assignment (represented here as a call expression) that takes as arguments an `identifier` and a list of pairs (value, label); it assigns to the identifier the value corresponding to the label of the block preceding the current one at run time;
- every basic block ends with a terminator which is a control flow-altering instruction that specifies which block to execute after termination of the current one.

```
function    = blocks:block*;
block       = label:identifier x phi_nodes:phi* x
              instructions:instruction* x end;
phi         = call;
instruction = load + store + call;
load        = identifier;
store       = name:identifier + value:expression;
end         = jump;
jump        = label:identifier;
```

Figure 7: Simplified Newgen definitions of the LLVM IR

Applying SPIRE to LLVM IR is, as illustrated above with PIPS, achieved in three steps, yielding the SPIREd parallel extension of the LLVM sequential IR provided in Figure 8.

- An `execution` attribute is added to `function` and `block`: a parallel basic block sees all its instructions launched in parallel (in a fork/join manner), while all the blocks of a parallel function are seen as parallel tasks to be executed concurrently.
- A `synchronization` attribute is added to `instruction`; hence, an instruction can be annotated with `spawn`, `barrier` or `atomic` synchronization attributes. When one wants to deal with a sequence of instructions, this sequence is first englobed in a `block` to whom`synchronization` is added.
- `send` and `recv` functions for handling data distribution are also seen as intrinsic. Moreover, pgas variables are introduced

in LLVM IR by enriching the format of `identifiers` with `location` information. One-sided communications are represented by adding `expression` to `load` instruction.

```
function′     = function x execution;
block′        = block x execution x synchronization;
instruction′  = instruction x synchronization;
load′         = load x expression;
store′        = store x expression;
identifier′   = identifier x location;
```

Figure 8: SPIRE (LLVM IR)

The use of SPIRE on the LLVM IR is not able to express parallel loops as easily as was the case on PIPS IR. Indeed, the notion of a loop does not always exist in the definition of IRs based on control-flow graphs, including LLVM; it is an attribute of some of its nodes, which has to be added later on by a loop-detection program analysis phase. Of course, such analysis could also be applied on the SPIRE-derived IR, to recover this information.

5.2 SPIRE Application to WHIRL

WHIRL (Winning Hierarchical Intermediate Representation Language) is the IR used in multiple compilers derived from SGI MIPS Pro compiler, such as the Open64 and PathScale compilers. Open64 has multiple branches, developed by Tensilica, Tsinghua University or Berkeley; the University of Houston offers its own open-source compiler, OpenUH. Currently, SPIRE(WHIRL) is being implemented in OpenUH, but hopefully all these branches will eventually use this extension in order to build clean and powerful parallel optimizations and transformations. Like PIPS IR, WHIRL is a hierarchical AST with 5 main levels: very high, high, mid, low, and very low WHIRL. Each level is adapted to some kinds of optimization and represents an intermediate interface among all the front-end and back-end components. In our work, we extend the front-end component and thus the very high WHIRL (VHWHIRL).

Figure 9 provides the definition of a significant subset of the sequential WHIRL described in [2], using Newgen. There, for instance, every tree is represented by a function entry node that contains a block node that contains the body of the function. A basic block is a list of subtrees or statements that can be loops, calls... The rest of the specification is rather straightforward.

```
function  = body:block;
block     = statements:statement*;
statement = doloop + call + dload + dstore ;
doloop    = index:symbol x initialize:dstore x
            cond:expression x step:dstore x block;
call      = fname:symbol x params:parameter*;
parameter = dload + iload;
symbol    = name:string x type x initial:value;
dload     = name:symbol;
dstore    = name:symbol x value:expression;
```

Figure 9: Simplified Newgen definitions of WHIRL

Applying the 3-step SPIRE process to WHIRL yields its SPIREd parallel extension provided in Figure 10.

- An `execution` attribute is added to `block` and `doloop`: a parallel basic block sees all its statements launched in parallel, while all the iterations of a parallel doloop are to be executed concurrently;
- A `synchronization` attribute is added to every type of statement such as `call` that can be annotated with `spawn`, `barrier` or `atomic` synchronization attributes. We could proceed by creating a new `instruction` node in WHIRL, as the union of all types of statements and the child of `statement`, and adding then `synchronization` only once to `statement`, as done for SPIRE(PIPS IR). However, for pragmatic reasons, we prefer to add `synchronization` on each type of statement rather than adding it once to a new node `instruction` in order to not change the compiler everywhere to adapt sequential passes to the introduction of this new node. Since WHIRL provides `intrinsic_call` nodes, we handle SPIRE events API for handling point-to-point synchronization with such nodes.
- `send` and `recv` functions are also intrinsic. Coarrays or pgas variables, in WHIRL, extend the `symbol` set while one-sided communications are represented by adding `expression` to `load` instruction.

```
block′    = block x execution x synchronization;
doloop′   = doloop x execution x synchronization;
call′     = call x synchronization;
dload′    = dload x synchronization x expression;
dstore′   = dstore x synchronization x expression;
symbol′   = symbol x location;
```

Figure 10: SPIRE(WHIRL)

5.3 SPIRE Performance Assessment

Parallelization. To assess the ability of the SPIRE methodology to design parallel IRs with enough expressivity for parallelism, we used it in PIPS, by upgrading its implementation so that it can handle the constructs described in Section 3. We have used ***"bad english this parallelism-enabled of PIPS" for the implementation of a new task parallelization algorithm [26]. It automatically generates both OpenMP and MPI code from the same parallel IR. We gathered performance data related to SPIRE-based parallelization on four well-known C scientific benchmarks, targeting both shared and distributed memory architectures: the image and signal processing benchmarks Harris [34] and ABF [17], the SPEC2001 benchmark equake [8] and the NAS parallel benchmark IS [32]. Our performance results exhibit significant speedups (see [26]) against the sequential versions of these programs. This experimental work of automatic parallelization suggests thus that the SPIRE methodology is able to provide parallel IRs that encode inherent parallelism, and is thus well adapted to the design of parallel target formats for the efficient parallelization of scientific applications on both shared and distributed memory systems.

One-Sided Communication Optimization. After the previous experiment, which provided promising global performance data for SPIRE, we decided to look in detail at a particular optimization, namely OpenSHMEM one-sided communication optimization, via the middle-end optimization layer of LLVM. SPIRE suggests to represent the one-sided communication primitives of OpenSHMEM in LLVM IR by memory load/store operations. We implemented the `location` domain as an annotation on the IR nodes, using LLVM metadata. A metadata in LLVM is a string used as an annotation on the LLVM IR nodes. After activating the middle-end optimizations of LLVM and Polly, we programmed the backend of LLVM to generate back the one-sided put/get function calls to be executed by the OpenSHMEM runtime. This two-step transformation process allows LLVM to identify and optimize communications in OpenSHMEM, seen as "simple" load/store operations.

As a specific case study, we show the impact of two optimizations: (1) applying loop tiling, a classical sequential optimization

already available in Polly, on a SPIRE-encoded OpenSHMEM program – the idea is to break down communications into successive chunk transfers using loop tiling in order to eliminate some of the runtime overhead of dynamic buffer management, –, and (2) performing the reverse transformation, i.e., communication vectorization, which transforms loops of load/store operations automatically into bulk put/get communications. For this later, we adapted the existing `LoopIdiomRecognize` LLVM pass that transforms simple loops into a non-loop form to work on load/store operations that have the same RMA metadata (remote PE): put/get operations are generated in lieu of `memcpy` intrinsics.

In our experiment, we used as microbenchmark [1] a `shmem_put` operation between a pair of processes, while varying the size of the transmitted array. We ran it on the Stampede supercomputing system at Texas Advanced Computing Center (TACC), under the OpenSHMEM implementation in MVAPICH2-X [21] version 2.0b. We compiled with LLVM-3.5.0, sporting the same version of Polly. Figure 11 (a) shows the impact of communication vectorization, which reduces both message startup time and latency, thanks to our modified `LoopIdiomRecognize` transformation. Figure 11 (b) shows that the unmodified LLVM middle-end optimizer was able to improve communication operations via tiling. This is particularly visible for large messages (≥ 2048), since we used a tile size of 2048 and the fact that MVAPICH2-X implementation already provides optimizations for small and medium message sizes. Had we not used our SPIRE encoding, LLVM (and any other compiler) would not have been able to apply these two optimizations, since it would have considered blindly the put operation as a function call with no particular semantic.

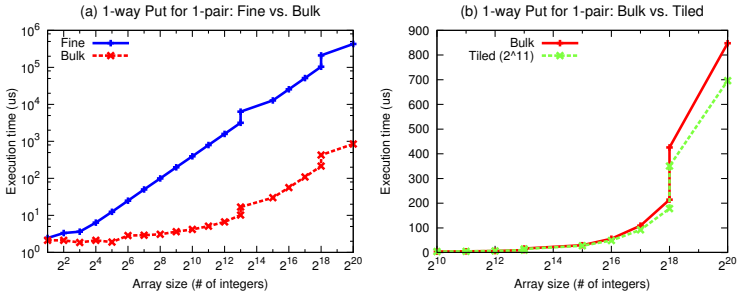

Figure 11: Performance comparison for communication vectorization and loop tiling in OpenSHMEM on Stampede using 2 nodes

6. Related Work

Syntactic approaches to parallelism expression use abstract syntax tree nodes, while adding specific parallel built-in functions. For instance, the IR of the implementation of OpenMP in GCC (GOMP) [31] extends its three-address representation, GIMPLE [30]. The OpenMP parallel directives are replaced by specific built-ins in low- and high-level GIMPLE, and additional nodes in high-level GIMPLE, such as the `__sync_fetch_and_add` built-in function for an atomic memory access addition. Similarly, Sarkar and Zhao introduce the high-level parallel IR HPIR [38] that decomposes Habanero-Java programs into region syntax trees, while maintaining additional data structures on the side: region control-flow graphs and region dictionaries. New syntax tree nodes are introduced: `AsyncRegionEntry` and `AsyncRegionExit` delimit tasks, while `FinishRegionEntry` and `FinishRegionExit` can be used in parallel sections. SPIRE borrows some of the ideas used in GOMP or HPIR, but frames them in more structured settings while trying to be more language-neutral. In particular, we try to minimize the number of additional built-in functions, which have the drawback of hiding the abstract high-level structure of parallelism and affecting compiler optimization passes. Moreover, we focus on extending existing AST nodes rather than adding new ones (such as in HPIR) in order not to fatten the IR and avoid redundant analyses and transformations on the same basic constructs.

InsPIRe is the parallel IR at the core of the source-to-source Insieme compiler [20] for C, C++, OpenMP, MPI and OpenCL. Parallel constructs are encoded using built-ins. SPIRE intends to also cover source-to-source optimization. It could have been applied to Insieme sequential components, parallel constructs being defined as extensions of the sequential abstract syntax tree nodes of InsPIRe instead of using built-ins such as `spawn` and `mergeAll`.

Turning now to mid-level intermediate representations, many systems rely on graph structures for representing sequential code, and extend them for parallelism. The Hierarchical Task Graph [16] represents the program control flow. The hierarchy exposes the loop nesting structure; at each loop nesting level, the loop body is hierarchically represented as a single node that embeds a subgraph that has control and data dependence information associated with it. SPIRE is able to represent both structured and unstructured control-flow dependence, thus enabling recursively-defined optimization techniques to be applied easily. The hierarchical nature of underlying sequential IRs can be leveraged, via SPIRE, to their parallel extensions; this feature is used in the PIPS case addressed below.

A stream graph [10] is a dataflow representation introduced specifically for streaming languages. Nodes represent data reorganization and processing operations between streams, and edges, communications between nodes. Each time a node is fired, it consumes a fixed number of elements of its inputs and produces a fixed number of elements on its outputs. Streaming can be handled in SPIRE using point-to-point synchronization, while SPIRE also provides support for both data and control dependence information.

The OSCAR Fortran Compiler [23] partitions programs into macro-task graphs (MTG), where vertices represent macro-tasks of three kinds, namely basic, repetition and subroutine blocks; a macro-flow graph is generated to represent data and control dependences on these macro-tasks. The parallel program graph (PPDG) [35] extends the program dependence graph [15], where vertices represent blocks of statements and edges, essential control or data dependences; *mgoto* control edges are added to represent task creation occurrences, and synchronization edges, to impose ordering on tasks. Like MTG and PPDG, SPIRE adopts an extension approach to "parallelize" existing sequential intermediate representations; our paper shows that this can be defined as a general mechanism for parallel IR definitions and provides a formal specification of this concept.

The LLVM compiler supports OpenMP but lowers all its pragmas at the front-end phase (in Clang) to runtime calls. In this work we added specific support for the one-sided operations of OpenSHMEM in LLVM, via load/store constructs of LLVM IR. This makes it possible to apply seamlessly LLVM transformations such as loop tiling and communication vectorization to OpenSHMEM programs (see Section 5).

7. Conclusion

SPIRE is a new 3-step methodology for the design of parallel language intermediate representations (IR); it maps any sequential IR used in compilation platforms to a parallel IR. This extension process introduces (1) a parallel execution attribute for each group of statements, (2) a high-level synchronization attribute on each statement and an API for low-level synchronization events, and (3) data location on processes together with two built-ins for implementing communications in message-passing memory systems. The formal semantics of SPIRE transformations are specified using a two-tiered approach: a small-step operational semantics for its base parallel concepts and a rewriting mechanism for high-level constructs.

The SPIRE methodology is presented via a use case, the IR of PIPS, a source-to-source compilation infrastructure for Fortran and C. We illustrate the generality of our approach by showing how SPIRE can be used to represent the constructs of eight parallel languages and libraries such as OpenMP, MPI and OpenSHMEM, and to extend LLVM IR and WHIRL. We provide experimental elements to validate SPIRE via its implementation in PIPS, for parallel code generation, and LLVM, for OpenSHMEM one-sided communication optimization.

Future work will address the use of SPIRE formal semantics to prove the correctness of optimizations performed on parallel programs. Besides loop tiling and communication vectorization, we plan to apply and adapt all transformations performed by LLVM to OpenSHMEM programs.

References

[1] HPCTools PGAS-Microbenchmarks. `https://github.com/uhhpctools/pgas-microbench`.

[2] *Open64 Compiler: WHIRL Intermediate Representation*. `www.mcs.anl.gov/OpenAD/open64A.pdf`, 2007.

[3] *MPI: A Message-Passing Interface Standard*. `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`, Sep 2012.

[4] OpenSHMEM Application Programming Interface (version 1.0). `http://upc.gwu.edu/documentation.html`, 2012.

[5] *X10 Language Specification*, Feb. 2013. URL `http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf`.

[6] *OpenMP Application Program Interface*. `http://www.openmp.org/mp-documents/OpenMP_4.0_RC2.pdf`, Mar. 2013.

[7] *Chapel Language Specification 0.796*, Oct 21, 2010. URL `http://chapel.cray.com/spec/spec-0.796.pdf`.

[8] H. Bao, J. Bielak, O. Ghattas, L. F. Kallivokas, D. R. O'Hallaron, J. R. Shewchuk, and J. Xu. Large-scale Simulation of Elastic Wave Propagation in Heterogeneous Media on Parallel Computers. *Comp. Methods in Applied Mech. and Eng.*, 152(1–2):85–102, 1998.

[9] V. Cavé, J. Zhao, and V. Sarkar. Habanero-Java: the New Adventures of Old X10. In *9th International Conference on the Principles and Practice of Programming in Java (PPPJ)*, Aug 2011.

[10] Y. Choi, Y. Lin, N. Chong, S. Mahlke, and T. Mudge. Stream Compilation for Real-Time Embedded Multicore Systems. In *Proc. of the 7th annual IEEE/ACM Int. Symposium on Code Generation and Optimization*, CGO '09, pages 210–220, Washington, DC, USA, 2009.

[11] F. Coelho, P. Jouvelot, C. Ancourt, and F. Irigoin. Data and Process Abstraction in PIPS Internal Representation. In F. Bouchez, S. Hack, and E. Visser, editors, *Proceedings of the Workshop on Intermediate Representations*, pages 77–84, 2011.

[12] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct. 1991.

[13] E. W. Dijkstra. Cooperating Sequential Processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[14] E. W. Dijkstra. Re: "Formal Derivation of Strongly Correct Parallel Programs" by A. van Lamsweerde and M. Sintzoff. circulated privately, 1977.

[15] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph And Its Use In Optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987. ISSN 0164-0925.

[16] M. Girkar and C. D. Polychronopoulos. Automatic Extraction of Functional Parallelism from Ordinary Programs. *IEEE Trans. Parallel Distrib. Syst.*, 3:166–178, Mar 1992. ISSN 1045-9219.

[17] L. Griffiths. A Simple Adaptive Algorithm for Real-Time Processing in Antenna Arrays. *Proceedings of the IEEE*, 57:1696 – 1704, 1969.

[18] T. Grosser, A. Groesslinger, and C. Lengauer. Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[19] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical Interprocedural Parallelization: An Overview of the PIPS Project. In *ICS*, pages 244–251, 1991.

[20] H. Jordan, S. Pellegrini, P. Thoman, K. Kofler, and T. Fahringer. INSPIRE: The Insieme Parallel Intermediate Representation. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 7–18, Piscataway, NJ, USA, 2013. IEEE Press.

[21] J. Jose, K. Kandalla, M. Luo, and D. Panda. Supporting Hybrid MPI and OpenSHMEM over InfiniBand: Design and Performance Evaluation. In *41st International Conference on Parallel Processing (ICPP)*, Sept 2012.

[22] P. Jouvelot and R. Triolet. Newgen: A Language Independent Program Generator. Technical report, CRI/A-191, MINES ParisTech, Jul 1989.

[23] H. Kasahara, H. Honda, A. Mogi, A. Ogura, K. Fujiwara, and S. Narita. A Multi-Grain Parallelizing Compilation Scheme for OSCAR (Optimally Scheduled Advanced Multiprocessor). In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 283–297, London, UK, 1992.

[24] D. Khaldi. *Automatic Resource-Constrained Static Task Parallelization: A Generic Approach*. PhD thesis, MINES ParisTech, Nov. 2013.

[25] D. Khaldi, P. Jouvelot, C. Ancourt, and F. Irigoin. Task Parallelism and Data Distribution: An Overview of Explicit Parallel Programming Languages. In H. Kasahara and K. Kimura, editors, *LCPC*, volume 7760 of *Lecture Notes in Computer Science*, pages 174–189, 2012.

[26] D. Khaldi, P. Jouvelot, and C. Ancourt. Parallelizing with BDSC, a Resource-Constrained Scheduling Algorithm for Shared and Distributed Memory Systems. *Parallel Computing*, 41(0):66 – 89, 2015.

[27] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[28] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng. OpenUH: An Optimizing, Portable OpenMP Compiler. *Concurr. Comput. : Pract. Exper.*, 19(18):2317–2332, Dec. 2007.

[29] *The LLVM Reference Manual (Version 3.7)*. The LLVM Development Team, Feb. 2015.

[30] J. Merrill. GENERIC and GIMPLE: a New Tree Representation for Entire Functions. In *GCC Developers Summit*, pages 171–180, 2003.

[31] D. Novillo. OpenMP and Automatic Parallelization in GCC. In *the Proceedings of the GCC Developers Summit*, Jun 2006.

[32] NPB. NAS Parallel Benchmarks. `http://www.nas.nasa.gov/publications/npb.html`.

[33] R. W. Numrich and J. Reid. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17:1–31, Aug 1998.

[34] T. Saidani, L. Lacassagne, J. Falcou, C. Tadonki, and S. Bouaziz. Parallelization Schemes for Memory Optimization on the Cell Processor: A Case Study on the Harris Corner Detector. In P. Stenström, editor, *Transactions on High-Performance Embedded Architectures and Compilers III*, volume 6590 of *LNCS*, pages 177–200. 2011.

[35] V. Sarkar and B. Simons. Parallel Program Graphs and their Classification. In U. Banerjee, D. Gelernter, A. Nicolau, and D. A. Padua, editors, *LCPC*, volume 768 of *Lecture Notes in Computer Science*, pages 633–655. Springer, 1993. ISBN 3-540-57659-2.

[36] J. Stanier and D. Watson. Intermediate Representations in Imperative Compilers: A Survey. *ACM Comp. Surv.*, 45(3):26:1– 27, July 2013.

[37] *Cilk 5.4.6 Reference Manual*. Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science, Nov. 2001.

[38] J. Zhao and V. Sarkar. Intermediate Language Extensions for Parallelism. In *Proc. of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, pages 329–340, New York, NY, USA, 2011. ACM.