The Incremental Design of Parallel Compiler Intermediate Representations using SPIRE

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 the OpenCL, Cilk, OpenMP, X10, Habanero-Java, CAF and Chapel parallel programming languages and MPI and OpenSHMEM libraries 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 successfully performs the encoding of control and/or data flow information. HPIR [34], PLASMA [29] or InsPIRe [18] are instances that operate at a high abstraction level, while the hierarchical task, stream or program dependence graphs (we survey these notions in Section 2) are better suited to graph-based approaches. Yet many more existing compiler frameworks use traditional representations for sequential-only programs [31], 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.

The main practical motivation of our proposal is to preserve the many years of development efforts invested in huge compiler plat-
forms such as GCC (more than 7 million SLOC), LLVM [25] (more than 1 million SLOC), PIPS [17] (600,000 SLOC), OpenUH [24] (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 compiler projects 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) [9] 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, OpenCL 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 [17], LLVM [25] and OpenUH [24].
- 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 OpenUH compiler, is being performed to validate SPIRE for the parsing of the parallel languages CAF and OpenMP.

After this introduction, we survey existing parallel IRs in Section 2. We describe our use-case sequential IR, part of the PIPS compilation framework, in Section 3. Our parallel extension proposal, SPIRE, is introduced in Section 4, where we also show how simple illustrative examples written in OpenCL, Cilk, OpenMP, X10, Habanero-Java, CAF, Chapel, MPI and OpenSHMEM can be easily represented within SPIRE. The formal operational semantics of SPIRE is given in Section 5. Section 6 illustrates the generality of SPIRE by applying it on LLVM and WHIRL. We discuss future work and conclude in Section 7.

2. 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) [27] extends its three-address representation, GIMPLE [26]. 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 HIRP [34] 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 HIRP, 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 HIRP) in order not to fatten the IR and avoid redundant analyses and transformations on the same basic constructs.

PLASMA is a programming framework for heterogeneous SIMD systems, with an IR [29] that abstracts data parallelism and vector instructions. It provides specific operators such as add on vectors and special instructions such as reduce and par. While PLASMA abstracts SIMD implementation and compilation concepts for SIMD accelerators, SPIRE is more architecture-independent and also covers control parallelism.

InsPRe is the parallel IR at the core of the source-to-source Insieme compiler [18] 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 InsPRe 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 [8] is a dataflow representation introduced specifically for streaming languages. Nodes represent data organization 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 its point-to-point synchronization primitives, while SPIRE also provides support for both data and control dependence information.

The OSCAR Fortran Compiler [20] 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) [30] extends the program dependence graph [15], where
vertices represent blocks of statements and edges, essential control or data dependences; \texttt{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.

3. 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 [17], 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 6.1 illustrates two other applications of SPIRE, on LLVM [25] and WHIRL [1].

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 [19], 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 \texttt{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 "\texttt{:=}" function. The call set is not defined here.

Instructions are included within statements, which are members of a cartesian product set (\times) 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 \texttt{declarations} here; all named objects such as user variables or built-in functions in PIPS are members of the \texttt{entity} set (the \texttt{value} set denotes constants while the "\texttt{x}" 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, i.e., a succession 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 \texttt{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.

4. 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 tech-

igues 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. Note that, in Section 6.1, using LLVM and WHIRL, we show that our approach is general enough to be adapted to other IRs.

4.1 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 [22], summarizes the main characteristics of nine recent and widely used parallel languages and libraries: Chapel, OpenCL, X10, Habanero-Java, OpenMP, Cilk, 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.

4.2 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:
Synchronization

<table>
<thead>
<tr>
<th>Language/Library</th>
<th>Execution</th>
<th>Synchronization</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapel (Cray)</td>
<td>forall coforall cobegin</td>
<td>reduce begin sync sync atomic sync</td>
<td>PGAS (Locates) on</td>
</tr>
<tr>
<td>OpenCL</td>
<td>EnqueueND-RangeKernel</td>
<td>— EnqueueTask Finish EnqueueBarrier atom_add, ...</td>
<td>events Distributed ReadBuffer WriteBuffer</td>
</tr>
<tr>
<td>X10 (IBM)</td>
<td>foreach</td>
<td>reduce async future finish atomic force next force</td>
<td>PGAS (Places) at</td>
</tr>
<tr>
<td>Habanero-Java (Rice)</td>
<td>foreach</td>
<td>accumulator async future finish atomic isolated get next get</td>
<td>PGAS (Places) at</td>
</tr>
<tr>
<td>OpenMP</td>
<td>omp for omp sections omp reduction omp task omp section omp taskwait omp barrier omp critical omp atomic —</td>
<td>Shared private, shared...</td>
<td></td>
</tr>
<tr>
<td>Cilk (MIT)</td>
<td>implicit</td>
<td>co_reduce — spawn sync cilk_lock —</td>
<td>Shared —</td>
</tr>
<tr>
<td>CAF (Cray)</td>
<td>implicit</td>
<td>co_reduce — sync all sync images critical end critical event post event wait</td>
<td>PGAS (Images) []</td>
</tr>
<tr>
<td>OpenSHMEM (UH &amp; ORNL)</td>
<td>start_pes shmem_sum shmem_pax ... — shmem_barrier_all shmem_set_lock shmem_clear_lock shmem_wait shmem_wait_until</td>
<td>PGAS (PEs) shmem_put shmem_get</td>
<td></td>
</tr>
<tr>
<td>MPI</td>
<td>MPI_Gut</td>
<td>MPI_Reduce MPI_Spawn MPI_Finalize MPI_BARRIER — — —</td>
<td>Distributed MPI_Send MPI_Recv...</td>
</tr>
<tr>
<td>SPIRE</td>
<td>parallel</td>
<td>reduced spawn barrier atomic signal wait</td>
<td>Shared, Distributed, PGAS send, recv, location</td>
</tr>
</tbody>
</table>

Table 1. Mapping of SPIRE to parallel languages constructs (implicit means the compiler will insert CAF_INIT to launch the images)

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:

\[
\text{execution} = \text{sequential:unit + parallel:unit + reduced:unit};
\]

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.

For instance, a parallel execution construct can be used to represent data parallelism on GPUs, when expressed via the OpenCL clEnqueueNDRangeKernel function (see Figure 2).

This function call could be encoded within the PIPS parallel IR as a parallel loop, each iteration executing the kernel function as a separate task, receiving the proper index value as an argument. In this paper, a task is a statement, to be executed by a thread or a process.

\[
global\_work\_size[0] = n; 
err = clEnqueueNDRangeKernel(cmd_queue, 
kernel, 1, NULL, global_work_size, 
NULL, 0, NULL, NULL);
\]

Figure 2: OpenCL parallel loop, executing \( n \) kernels

An another example, in the left side of Figure 3, 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.

\[
\begin{align*}
\text{forall } i \in 1..n & \text{ do } \text{t}[i] = i; \\
\text{var } (\text{sumVal}) & = + \text{reduce} \\
(1 \ldots n) f(i), & 1..n) \\
\text{sumVal} = \text{sumVal} + f(i), & \text{reduced})
\end{align*}
\]

Figure 3: SPIRE version of forall and reduce in Chapel

4.3 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.

4.3.1 Collective Synchronization

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

\[
\text{statement’ = statement x synchronization};
\]

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:

\[
\text{synchronization} = \text{none:unit + spawn:entity + barrier:unit + atomic:reference}.
\]

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 cor-responding 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;
• Barrier specifies that all the child threads spawned by the execution of \( S \) are suspended before exiting until they are all finished — an OpenCL example illustrating spawn (clEnqueueTask) and barrier (clEnqueueBarrier) is provided in Figure 4;

```c
Cilk_lock(l);
Cilk_lockvar l;
...
Cilk_lock(l);
x[index[i]] += f(i);
Cilk_unlock(l);
```

Figure 5: Cilk and OpenMP variants of atomic statements

4.3.2 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:

```c
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 (see Figure 6) 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 signaled when the task is completed and waited by the force method.

```c
future<int> Fi = future{foo()};
int i = Fi.force();
```

Figure 6: X10 variant of task synchronization

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 7, left). The async phaser keyword can be replaced by spawn. In this example, the next statement is equivalent to the following sequence:

```c
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.

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

Figure 7: 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.

4.4 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 [33] memory model, which appears in languages such as UPC [10], 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.
4.4.1 Memory Information
SPIRE is able to handle all three memory models using specific memory information, namely private, shared and pgased. Each process has its own private memory; the address of a shared variable refers, within each thread, to the same physical memory location; pgased memory is distributed evenly among different processes. In this last case, e.g., in OpenSHMEM and CAF, all processes have their own view of pgased 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 pgased memory. This is done by adding a location domain to the storage domain of PIPS IR:

\[
\text{storage'} = \text{storage} \times \text{location};
\]

\[
\text{location} = \text{private:unit} + \text{shared:unit} + \text{pgased:expression};
\]

where the list of expressions in the pgased model is used to distinguish processes, logically distributed on a matrix. 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 pgased memory. First, in CAF, coarrays, which are extension of Fortran arrays, are pgased 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.

\[
\text{integer}(\text{len}=20) :: \text{dest}[*]
\]

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

\[
\text{dest} = (\text{int}*)\text{shmalloc(sizeof(int)} \times 20);
\]

4.4.2 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 \texttt{send} and \texttt{recv} blocking functions for implementing communication between processes:

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

Non-blocking communications can be easily implemented in SPIRE using the above primitives within spawned statements. Also, broadcast collective communications, such as defined in MPI, can be seen as wrappers around \texttt{send} and \texttt{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 \texttt{send} as body; otherwise (receiver), the broadcast is a \texttt{recv} function.

The MPI example in Figure 8 can be represented in SPIRE as a sequential loop with index \texttt{my_rank} of size iterations whose body spawns the MPI code from \texttt{MPI_Comm_size} to \texttt{MPI_Finalize}, using \texttt{my_rank} as process number. The communication of Variable \texttt{sum} from Process 1 to Process 0 can be handled with SPIRE \texttt{send/recv} functions.

4.4.3 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 \texttt{location} domain of entities, there is no need to gather again this information (see Section 5.2). 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:

\[
\text{shmem\_int\_put(dest, src, 20, pe)};
\]

can be represented in SPIRE by:

\[
\text{arraysection(dest, 20)} = \text{arraysection(src, 20)}
\]

if we suppose that the sequential IR provides an \texttt{arraysection} construct to represent subarrays and that the \texttt{location} of \texttt{dest} is \texttt{pgased([pe])}.

5. SPIRE Operational Semantics
The purpose of the formal definition described in this section is to provide a solid basis 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.

5.1 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 simpler, minimal sequential language, \texttt{Stmt}. Its syntax and the syntax of the set \texttt{Exp} of expressions \texttt{E} are given in Figure 9, where we assume that the set \texttt{Id} of identifiers \texttt{I} is given.

\[
S \in \text{Stmt} ::= \text{nop} \mid \text{I=E} \mid S_1;S_2 \mid \text{loop}(E,S)
\]

\[
E \in \text{Exp} ::= \text{I} \mid \text{E}_1 + \text{E}_2
\]
Sequential statements are: (1) \textit{nop} for no operation, (2) \textit{I=E} for an assignment of \textit{E} to \textit{I}, (3) \textit{S_1;S_2} for a sequence and (4) \textit{\texttt{loop}}(\textit{E,S}) for a while loop.

At the semantic level, a statement in \textit{Stmt} is a very simple memory transformer. A memory \( m \in \textit{Memory} \) is a mapping in \( \textit{Ide} \rightarrow \textit{Value} \), where values \( v \in \textit{Value} = N + \text{Bool} \) can either be integers \( n \in N \) or booleans \( b \in \text{Bool} \). The sequential operational semantics for \textit{Stmt}, expressed as transition rules over configurations \( \kappa \in \textit{Configuration} = \text{Memory} \times \textit{Stmt} \), is given in Figure 10; we assume that the program is syntax- and type-correct. A transition \((m,S) \rightarrow (m',S')\) 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. Rules 1 to 5 encode typical sequential small-step operational semantic rules for the sequential part of the core language. We assume that \( \xi \in \text{Exp} \rightarrow \text{Memory} \rightarrow \text{Value} \) is the usual function for expression evaluation.

\[
\begin{align*}
\text{Stmt} & : \quad \texttt{nop} | \texttt{I=E} | \texttt{S_1;S_2} | \texttt{\texttt{loop}}(E,S) | \\
\text{Proc} & : \quad \texttt{send}(I,I') | \texttt{recv}(I,I') \quad \text{for all processes} \quad I, I' \in \text{Proc}
\end{align*}
\]

\textbf{Figure 11: SPIRE (Stmt) and SPIRE (Ide) syntaxes}

\begin{align*}
\textbf{Figure 10: Stmt sequential transition rules} \\
\text{Stmt} & : \quad \texttt{nop} | \texttt{I=E} | \texttt{S_1;S_2} | \texttt{\texttt{loop}}(E,S) | \\
\text{Proc} & : \quad \texttt{send}(I,I') | \texttt{recv}(I,I') \quad \text{for all processes} \quad I, I' \in \text{Proc}
\end{align*}

### 5.2 SPIRE as a Language Transformer

\textbf{Syntax} At the syntactic level, SPIRE specifies how a grammar for a sequential language such as \textit{Stmt} is transformed, i.e., extended, with synchronized parallel statements. The grammar of \textit{SPIRE(Stmt)} in Figure 11 adds to the sequential statements of \textit{Stmt} (from now on, synchronized using the default none) new parallel statements: a task creation \texttt{spawn}, a termination \texttt{signal} and two \texttt{wait} operations on events or \texttt{send} and \texttt{recv} operations for communication. Synchronization atomic is defined via rewriting (see below). The statement \texttt{barrier} \((n)\), added here for specifying the multiple-step behavior of the \texttt{barrier} statement in the semantics, is not accessible to the programmer. Figure 7 provides the SPIRE representation of a program example.

Note that the grammar of \textit{Ide} is also extended to \textit{SPIRE(Ide)} in Figure 11 to add the possibility of performing a memory access to the pgased identifier \( i \) located on Process \( E \), noted \( I[E] \). This extension of \textit{Ide} naturally carries over to \textit{Exp}.

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

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

\[ S \in \textit{SPIRE(Stmt)} ::= \texttt{nop} | \texttt{I=E} | \texttt{S_1;S_2} | \texttt{\texttt{loop}}(E,S) | \]

\[ \texttt{spawn}(I,S) | \texttt{barrier}(S) | \texttt{barrier\_wait}(n) | \]

\[ \texttt{wait}(I) | \texttt{signal}(I) | \]

\[ \texttt{send}(I,I') | \texttt{recv}(I,I') \]

\[ I \in \textit{SPIRE(Ide)} ::= I | I[E] \]
scissial design decision, to ensure that sequential optimizations such as strength reduction over private expressions remain valid.

In Rule 9, `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 10 implements a rendezvous: a new process executed and, when all the child processes spawned by `n` have finished.

In Rules 12 and 13, `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 `new Event` 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 \( \pi(I) = n \) with \( n > 0 \), the resource can be used or the barrier can be crossed. In Rule 13, the current value `n'` of `I` is incremented; this is a non-blocking operation.

In Rule 14, `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 [21].

6. 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. We illustrate how SPIRE can be applied for two different widely used compilers, namely LLVM and Open64-based OpenUH, by extending their respective sequential IRs with minimal changes, thus providing support regarding the generality of our methodology.

6.1 SPIRE Application to LLVM IR

LLVM [25] (Low-Level Virtual Machine) is an open-source compilation framework that uses an intermediate representation in Static Single Assignment (SSA) [11] form. 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 13 provides the definition of a significant subset of the sequential LLVM IR described in [25], 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 \( \phi \) nodes and a list of instructions, and ends with a terminator instruction;
- \( \phi \) nodes, which are the key elements of SSA, are used to merge the values coming from multiple basic blocks. A \( \phi \) 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.

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 14.

- 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 engulfed in a block to whom a synchronization attribute 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.

Note that 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, and thus recover this information.

6.2 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 (VWHIRL). Figure 15 provides the definition of a significant subset of the sequential WHIRL described in [1], 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.

Applying the 3-step SPIRE process to WHIRL yields its SPIREd parallel extension provided in Figure 16.

- 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 se-
intermediate representations (IR) which maps any sequential IR 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-mal 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 nine parallel parallelism. In Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques, PACT ’13, pages 7–18, Piscataway, NJ, USA, 2013. IEEE Press.


