INSPIRE
The Insieme Parallel Intermediate Representation

Herbert Jordan, Simone Pellegrini, Peter Thoman, Klaus Kofler, and Thomas Fahringer
Institute of Computer Science, University of Innsbruck
Technikerstrasse 21a, 6020 Innsbruck, Austria
Email: {herbert,spellegrini,petert,klaus,tf}@dps.uibk.ac.at

Abstract—**Programming standards like OpenMP, OpenCL and MPI are frequently considered programming languages for developing parallel applications for their respective kind of architecture. Nevertheless, compilers treat them like ordinary APIs utilized by an otherwise sequential host language. Their parallel control flow remains hidden within opaque runtime library calls which are embedded within a sequential intermediate representation lacking the concepts of parallelism. Consequently, the tuning and coordination of parallelism is clearly beyond the scope of conventional optimizing compilers and hence left to the programmer or the runtime system.**

The main objective of the Insieme compiler is to overcome this limitation by utilizing INSPIRE, a unified, parallel, high-level intermediate representation. Instead of mapping parallel constructs and APIs to external routines, their behavior is modeled explicitly using a unified and fixed set of parallel language constructs. Making the parallel control flow accessible to the compiler lays the foundation for the development of reusable, static and dynamic analyses and transformations bridging the gap between a variety of parallel paradigms.

Within this paper we describe the structure of INSPIRE and elaborate the considerations which influenced its design. Furthermore, we demonstrate its expressiveness by illustrating the encoding of a variety of parallel language constructs and we evaluate its ability to preserve performance relevant aspects of input codes.

Keywords—Compiler, Intermediate Representation, Parallel Computation, High-level Program Analysis

I. Introduction

Fully utilizing the potential of contemporary and emerging parallel architectures involving shared as well as private memory segments, multi-core CPUs, GPUs and possibly additional acceleration hardware, is known to be a challenging task. In the past, to empower software developers to deal with the challenges at hand, specialized programming models were developed, resulting in the establishment of standardized APIs and language extensions following those proposals. OpenMP, Cilk, MPI and OpenCL are among those standards, covering a variety of architectures. In addition to this range of programming models, the growing complexity of architectures led to combinations of those technologies. Hybrid programming models [1] such as MPI / OpenMP and OpenMP / OpenCL were established to fully exploit the capabilities of latest hardware architectural trends.

Parallel codes can benefit from compiler supported optimizations [2]–[6]. However, unlike conventional instruction level manipulation, this kind of analyses and transformations are preferably performed at a higher level of abstraction, reflecting the coarse grained nature of the targeted parallel constructs. Consequently, designers of such optimization and tuning aids focus on source-to-source compilation infrastructures based on high-level intermediate languages[1] such as those offered by ROSE [7], or CIL [8]. The internal representations (IRs) of those systems are capable of modeling code fragments close to the original C, C++ or Fortran source. However, since parallel constructs are not native constructs within those input languages, they are not explicitly reflected in the resulting IRs. The parallel control flow remains hidden within directives or calls to opaque external libraries and is hence beyond the scope of the language specification the underlying IR and the associated utilities are based on. The occasionally rich, end-user oriented semantic of the potentially large number of parallel constructs is determined by the corresponding API specification. Clearly, all of those constructs are required to be interpreted correctly when analyzing parallel aspects of a code fragment. Approaches aiming to support hybrid solutions face the even harder problem of dealing with multiple API specifications, potentially based on deviating terminology, throughout their system.

In this paper we present INSPIRE, the Insieme parallel intermediate representation, as it is used within the Insieme compiler project [9]. INSPIRE is a formal intermediate language for modeling heterogeneous parallel applications using a single, unified, and small set of composable constructs. In particular, parallel aspects of the represented codes are modeled on a language level. Consequently, analysis and manipulation utilities built on top of INSPIRE consider the parallel aspects of the handled code fragments natively. Furthermore, since input codes written using different parallel programming languages are internally represented based on a unified set of primitives, analyses and transformations originally implemented for a specific language can be reused for others. Boundaries between parallel input languages are implicitly eradicated.

To the best of our knowledge our system is the first compiler platform realizing support for a variety of parallel programming languages by converting them into a unified, high-level, language independent, parallelism-aware, formal intermediate representation. The major contributions of our work include:

This research has been partially funded by the Austrian Research Promotion Agency under contract 834307 (AutoCore) and by the FWF Austrian Science Fund under contract I01079 (GEMSCLAIM) and contract TRP 237-N23 (BPEL-MP).

[1]we use the terms intermediate language and representation interchangeably

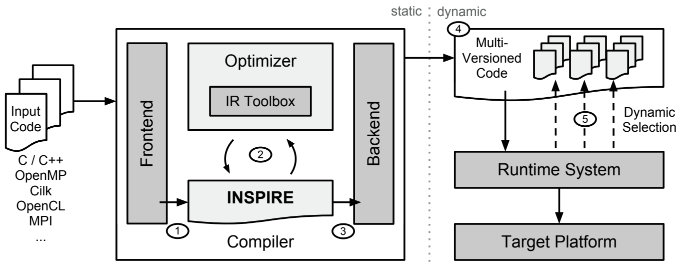

Fig. 1: Example setup using the Insieme infrastructure.

- the design of an extensible, high-level intermediate language comprising explicit parallel constructs to unify the representation of heterogeneous, parallel codes
- the modeling of OpenMP, Cilk, OpenCL and MPI primitives using a unified, closed set of intermediate language constructs
- the establishment of an extensible compiler infrastructure for (hybrid) parallel codes based on the separation of the description of parallelism from its implementation

II. Design Goals

A. The Insieme Project

The intermediate language presented within this paper has been developed to provide the foundation for the Insieme compiler project [9]. Consequently, its design has been heavily influenced by the objectives of the enclosing system. The Insieme project aims on establishing an infrastructure for researching parallel languages, program analyses, transformations, (auto-) tuning, and optimization solutions for computationally intensive real-world applications.

Figure 1 illustrates an example setup of the Insieme infrastructure for processing parallel codes. Input programs written using C/C++ based OpenMP, Cilk, OpenCL or MPI constructs are parsed using a Clang [10] based frontend and converted into the language independent IR presented within this paper (1). Within the compiler, research-specific analysis and transformations can be applied on the processed application (2). For this purpose a growing set of re-usable tools is offered. In the final phase of the compilation process, an exchangeable backend implementation converts the IR into source code targeting some sort of independent parallel runtime environment. The default implementation creates C/C++ code utilizing the framework established by the Insieme runtime system for processing parallel control flows (3). In this setup, besides conventional compiler based code optimizations, enriching the input program with additional dynamic tuning opportunities and meta-information derived from compile-time analyses is among the primary objectives of the compiler. A frequent use case of our system is based on the generation of multi-versioned code (4). For instance, problem size or hardware specific implementations of performance critical code regions may be introduced by the compiler. Based on the concrete scenario and meta-information on the variants forwarded from the compiler to the runtime system the proper version is selected dynamically during runtime (5). Beside the selection of code variants, the runtime system also deals with work-load scheduling, data distribution and resource allocation issues, all of which can be aided by information on the processed code gathered during compilation.

B. IR Requirements and Principles

All IRs of optimizing compilers should satisfy four major criteria to a certain degree within their specialized domain. They have to be

1) *Expressive* – the IR must be capable of expressing all relevant features of the input language as well as all aspects the optimizing compiler is intended to tune within the processed codes
2) *Analyzable* – the IR should be structured in a way such that analyses required for the optimization process can be (efficiently) conducted
3) *Transformable* – the IR design has to reflect the requirement of an optimizing compiler to efficiently manipulate the IR as well as to convert input codes into the IR and the IR into target code
4) *Extensible* – since compiler projects are running for several years, requirements on their IRs are likely to evolve over time. An IR should be adaptable to new requirements with a limited impact on the IR's implementation and related utilities

Several of those criteria are conflicting with each other. For instance, a very expressive, detailed IR like a complete C/C++ AST is harder to analyze than a more restricted IR [8]. To resolve some of the conflicts, we derived the following principles for the design of INSPIRE:

- *Explicit* – all important concepts are covered explicitly at a controllable level of abstraction to simplify analysis and allow transformations to affect those aspects; e.g. explicit parallelism and memory management
- *Unified* – constructs constituting the same meaning shall be expressed using identical means to foster the reusability of analyses and transformations; e.g. barriers or reductions originating from different input languages shall be represented using the same primitives
- *Simple* – IR constructs shall have a precise, non-overloaded interpretation to avoid dealing with special cases; e.g. for-loops should be restricted to their count-controlled interpretation
- *Modular* – the IR shall be separated into a fixed core component providing the means to easily define and manipulate extensions and a set of modular extensions modeling relevant concepts; extensions shall be handled within the intermediate language, not its implementation
- *Compact* – there shall be as few constructs as necessary

Based on those principles we developed and implemented INSPIRE as it is described in the following section.

III. The Representation

A. Basic Concepts

The design of our IR has been strongly inspired by functional programming languages and their powerful means for building flexible and reusable functionalities. This property is based on a simple core model determining the semantics of the program execution (lambda calculus), supporting functions as first-class citizens, the resulting power of functional composition and a type system including variables enabling the definition of generic implementations of functions. We have built our IR based on these concepts, thereby laying the foundation for the satisfaction of most of our design principles.

Beside the properties inherited from functional languages, the concept of abstract types and functions over those types has been inherited from formal *specification languages*. Within those languages, algebraic structures consisting of a set of abstract values (a type) and several operators defined on those can be used to specify the behavior of a program or system on a very high level. Within INSPIRE the same principle is used to abstract from low-level implementation details. In fact, all primitive types, several basic data structures and external library interfaces are modeled using abstract types. Also, by providing means for defining additional abstract types and operators, no language level modifications are required in case new types and operators have to be integrated.

1) Organization: The definition of INSPIRE is subdivided into two components. The *language core* covers a fixed set of primitives, including type, expression and statement constructs. It specifies rules for the deduction of types, the composition and the evaluation order of expressions, the processing of statements and the scope of variables. The core has been designed to be minimal and simple.

Based on the core's abilities, *extensions* may be defined. Extensions introduce new types and operators by either using abstract constructs or – whenever possible – by composing previously defined elements to obtain *derived constructs*. Neither requires the implementation of the IR to be altered. When using abstract constructs, they have to be interpreted correctly by IR utilities including the backend while derived components may be substituted by their definitions. Consequently, in order to keep the number of cases to be interpreted by utilities low, new abstract constructs should only be introduced if the desired behavior can not be expressed otherwise. In case constructs have to be added, the support for generic types and functions inherited from the functional roots facilitates the coverage of entire families of operators and types using a single construct.

2) Data Structure: Beside the expressive capabilities of our IR, an important goal of the design phase has been the facilitation of a simple and efficient implementation of the IR data structure, inspection utilities, and manipulation tools. Inspired by the simplicity of the self-contained structure of formal and functional languages, we extended this concept to our entire IR, resulting in a plain tree structure not containing any cross or back edges and reflecting the recursive definition of all our language constructs. Consequences of this design decision are discussed in detail within sections III-D and V.

The structure of the IR is focused solely on the representation of relevant semantic aspects of a program. However, in some cases, additional information like user defined hints on loop scheduling policies, results of analyses, or assertions on values may have to be stored somewhere. To provide an integrated means to record this kind of information, every IR node can be annotated with generic information. These annotations can be utilized for forwarding meta-information on constructs through various stages of the compiler or even to the runtime system.

B. Sequential Control Flow

Although ultimately focusing on parallel applications, the design of the constructs to be used to represent sequential control flow within the IR is of utmost importance for most design goals. Also, even within most sophisticated parallel applications, sequential constructs are utilized to describe the steps to be conducted by the involved processing units and to orchestrate the parallel code sections. Unfortunately, due to limitations on the number of pages, only an overview on the design of the sequential part of the IR can be covered[2].

1) Types: The set of all types $\mathbb{T}$ is defined as follows. Let $\mathbb{N}$ be the set of natural numbers including 0 and $\mathbb{I}$ be a set of identifiers. Let $i, i_1, \ldots, i_k \in \mathbb{I}$ be identifiers, $t, t_1, \ldots, t_k \in \mathbb{T}$ be types, $p_1, \ldots, p_l \in \mathbb{N} \cup \mathbb{I}$ be type parameters and $n_1, \ldots, n_k \in \mathbb{I}$ be names for $k, l \in \mathbb{N}$. Then, for all $x \in \mathbb{N} : 1 \leq x \leq k$,

$$i \langle t_1, \ldots, t_k, p_1, \ldots, p_l \rangle \tag{1}$$
$$struct\ \{n_1 : t_1, \ldots, n_k : t_k\} \tag{2}$$
$$union\ \{n_1 : t_1, \ldots, n_k : t_k\} \tag{3}$$
$$(t_1, \ldots, t_k) \rightarrow t \tag{4}$$
$$(t_1, \ldots, t_k) \Rightarrow t \tag{5}$$
$$'i \tag{6}$$
$$rec\ 'i_x : \{'i_1 = t_1, \ldots, 'i_k = t_k\} \tag{7}$$

are types as well. Note that (1), (2), (3) and (6) constitute base cases for this recursive definition whenever $k = l = 0$.

Types of form (1) support the introduction of *parametrized abstract types*. Primitive types including the boolean type $bool$[3], the signed 4-byte integer $int\langle 4\rangle$ or the double type $real\langle 8\rangle$ are represented within our IR using this construct. Also basic data structures including dynamically sized arrays of booleans ($array\langle bool\rangle$) or statically sized vectors of eight integers ($vector\langle int\langle 4\rangle, 8\rangle$) are based on this type construct.

The remaining type constructs support the construction of struct (2), union (3), function (4) and closure types (5). Type variables of the form (6) can be used to define generic types like $(\alpha) \rightarrow \alpha$ to type the identity function. For an easier distinction to other identifiers Greek letters like α and β are used instead of $'a$ and $'b$. The last construct (7) provides a means for defining recursive types like lists or tree structures. It binds the type variables $'i_1, \ldots, 'i_k$ to the types $t_1, \ldots, t_k$ to be used within the definition of types $t_1, \ldots, t_k$ for establishing recursive relations. The variable $'i_x$ selects the recursive type to be represented by the full construct. The support of multiple bound identifiers extends the expressiveness to mutually recursive types.

[2]We omit the few extensions required to model C++ constructs.

[3]If $k = l = 0$ we omit the trailing $\langle\rangle$.

2) Expressions: Let $\mathbb{V} = \{var(i:t) | i \in \mathbb{I} \wedge t \in \mathbb{T}\}$ be the set of variables where $var(i:t)$ denotes a variable i of type t. For conveniences we write i instead of $var(i:t)$ whenever its interpretation is clear from the context.

For the sequential control flow the set of expressions $\mathbb{E}$ is defined as follows. Let $v_* \in \mathbb{V}$ be variables[4], $i \in \mathbb{I}$ be an identifier, $n_* \in \mathbb{I}$ be names, $t_* \in \mathbb{T}$ be types, $e_* \in \mathbb{E}$ be expressions and s_* be statements for $k, l_* \in \mathbb{N}$. Then, for all $x \in \mathbb{N} : 1 \leq x \leq k$,

$$v \tag{1}$$
$$lit(i:t) \tag{2}$$
$$e(e_1, \ldots, e_k) \tag{3}$$
$$struct\,\{n_1 = e_1, \ldots, n_k = e_k\} \tag{4}$$
$$union\,\{n = e\} \tag{5}$$
$$\begin{aligned} &rec\ f_x\{\\ &\quad f_1 = (v_{11}, \ldots, v_{1l_1}) \rightarrow t_1\ s_1,\\ &\quad \ldots,\\ &\quad f_k = (v_{k1}, \ldots, v_{kl_k}) \rightarrow t_k\ s_k\\ &\} \end{aligned} \tag{6}$$
$$(v_1, \ldots, v_k) \Rightarrow e(e_1, \ldots, e_l) \tag{7}$$

are expressions as well.

Variables (1), *literals* (2) and *call* (3) expressions are required to model the data and control flow while the remaining four enable the construction of struct (4), union (5), function (6) and closure (7) values. Literals provide the means for the introduction of typed constants within the IR. Also, to introduce an abstract function/operator, a constant exhibiting a function type is used. Within *call* expressions (3), arguments are eagerly evaluated in an arbitrary order and passed by value. *Lambda* expressions (6) support the definition of (mutual) recursive functions by binding function definitions to variables ($f_1 \ldots f_k$). In case of a non-recursive function $rec\ f\{f = (v_1, \ldots, v_k) \rightarrow t\ s\}$, where f does not appear free in s, we use the short form $(v_1, \ldots, v_k) \rightarrow t\ s$. The bodies $s_1, \ldots, s_k$ of the function definitions must not exhibit free variables other than the associated parameters and the variables $f_1, \ldots, f_k$. Finally, *bind* expressions (7) construct a closure value processing a call expression $e(e_1, \ldots, e_l)$ upon invocation. Arguments $e_1, \ldots, e_l$ which are not parameters of the resulting closure $(v_1, \ldots, v_k)$ are captured from the surrounding context.

All expressions have an associated type which can be automatically deduced using inductive type deduction rules. An important feature of our IR is the support for generic function types. When invoking generic functions using a *call* expression, their types are instantiated according to the types of the passed arguments.

3) Statements: Beside every expression being a statement, eight additional statement constructs are supported in our IR: *compound*, *variable declaration*, *if*, *while*, *for*, *return*, *break* and *continue* statements.

The selected set of constructs reflects typical elements found within imperative languages. A *variable declaration* introduces a new variable whose scope is bound statically by the enclosing compound statement. Also, the IR core language offers two distinct constructs for condition-controlled *while* loops and count-controlled *for* loops. The execution of the latter must not be terminated by a $break$ or $return$ statement.

Justification for For-Loops: Offering two loop constructs (for and $while$) violates two of our design principles (*Unified* and *Compact*). After all, a single loop-forever construct as used within CIL [8] would suffice. However, for high-level optimization and tuning of codes, count-controlled loops are of major importance. Many techniques improving the cache utilization are based on loop transformations which are almost exclusively defined for for-loops. Even advanced techniques, like the polyhedral model [11], rely on code segments being structured using the available constructs.

4) Mutable State: So far, our IR consists solely of *pure* language features. Yet our input languages are imperative and require mutable state. Within our IR, this is modeled via an extension introducing the abstract generic type $ref\,\langle\alpha\rangle$, a variation of the concepts presented in [12]. A value of this type represents a reference to a memory location containing a value of type α. The following operators are defined on references:

Operator	Type	Description
$ref.var$	$(type\,\langle\alpha\rangle) \rightarrow ref\,\langle\alpha\rangle$	allocation of stack memory
$ref.new$	$(type\,\langle\alpha\rangle) \rightarrow ref\,\langle\alpha\rangle$	allocation of heap memory
$ref.del$	$(ref\,\langle\alpha\rangle) \rightarrow unit$	release of heap memory
$ref.deref$	$(ref\,\langle\alpha\rangle) \rightarrow \alpha$	read value from location
$ref.assign$	$(ref\,\langle\alpha\rangle\,, \alpha) \rightarrow unit$	update value in location
$ref.null$	$(type\,\langle\alpha\rangle) \rightarrow ref\,\langle\alpha\rangle$	get an uninitialized reference
$ref.isNull$	$(ref\,\langle\alpha\rangle) \rightarrow bool$	test validity of reference

Only the first two operators are capable of allocating memory – either bound to the current scope or manually released. The type $type\,\langle\alpha\rangle$ is used as a meta type to specify the type of the memory location to be allocated. The operator $ref.assign$ is the only operator capable of mutating state. As in other functional programming languages, its return type $unit$ reflects its sole purpose of causing a side effect.

References eliminate the requirement of distinguishing between r- and l-values. Within C the type *bool* may be assigned to a mutable memory location (C-variable) as well as to a temporary result obtained from an operation. In our IR, those two cases are distinguishable by type. The first is represented using an expression of type $ref\,\langle bool\rangle$ while the second case is represented by a value of type *bool*.

Furthermore, the introduction of reference types enables arguments to be passed or captured by reference. Also, pointers can be modeled using nested references. For instance, the type $ref\,\langle ref\,\langle bool\rangle\rangle$ models the concept of a mutable pointer to a boolean.

5) Arrays: An important construct frequently targeted by performance optimizing techniques are arrays and accesses to those. In INSPIRE arrays are modeled uniformly using the abstract generic type family $array\,\langle\alpha\rangle$. A value of type $array\,\langle\alpha\rangle$ represents a collection of data elements of the generic type α. The following operations are defined on arrays:

Operator	Type
$array.create$	$(type\,\langle\alpha\rangle\,, uint\,\langle 8\rangle) \rightarrow array\,\langle\alpha\rangle$
$array.elem$	$(array\,\langle\alpha\rangle\,, uint\,\langle 8\rangle) \rightarrow \alpha$
$array.ref.elem$	$(ref\,\langle array\,\langle\alpha\rangle\rangle\,, uint\,\langle 8\rangle) \rightarrow ref\,\langle\alpha\rangle$

[4]For brevity we use the notation x_* to denote the list of elements x with any or no subscript encountered within the following definition.

The $array.create$ operator creates an (immutable) array value containing the given number of undefined elements of type α and the operator $array.elem$ can be utilized to obtain the value at a given index position. In general the result of an application of the create operator is assigned to a reference referencing a memory location capable of holding an element of an array type, e.g. a variable of type $ref\,\langle array\,\langle bool\rangle\rangle$. To access an element within such a mutable array the $array.ref.elem$ operator is offered, obtaining a reference to the addressed element which then might be read (using $ref.deref$) or updated (using $ref.assign$).

Higher dimensional arrays are constructed by nesting one-dimensional arrays. Also, further extensions covering additional containers including statically sized vectors or lists have been defined.

C. Parallel Control Flow

The parallel model used within INSPIRE has been designed to cover the full flexibility of common parallel languages using a concise formalism. It is based on recursively nested *thread groups*. Figure 2 illustrates a snippet of the execution of a parallel application. On the top level, a thread group consisting of two threads is shown. The first of these threads spawns an inner thread group consisting of three threads while the second thread starts a sub-thread group which spawns two additional groups, each consisting of a single thread.

Within this context, we are using the term *thread* to refer to an arbitrary entity capable of processing a sequential control flow. This definition covers standard OS-level threads as well as OpenMP threads, Cilk tasks, OpenCL work items and MPI processes. INPSIRE's parallel model does not define any specific relation between its internal *threads*, OS-level threads, processes, HW-threads or other processing entities. This flexibility is exploited by the backend and runtime system, allowing them to implement IR threads using light-weight tasks which are flexibly scheduled on the available computational resources.

1) Jobs: Each thread group cooperatively processes a *job*. The creation of a job is supported by a job expression of the form

$$job\,[e_l, e_u]\,(v_1 = e_1, \ldots, v_k = e_k)\ \ f$$

where $e_l, e_u, e_1, \ldots, e_k \in \mathbb{E}$ are expressions, $v_1, \ldots, v_k \in \mathbb{V}$ are variables, $k \in \mathbb{N}$, and $f \in \mathbb{E}$ is a function of type $() \Rightarrow unit$. The value constructed by a job expression is of type job.

The function f specifies the operations to be conducted by each of the threads of a thread group when processing the resulting job. The expressions e_u and e_l define lower and upper boundaries for the number of threads required for processing the job. The concrete number is selected dynamically by the execution environment based on the available resources and according to some scheduling strategy. The variables $v_1, \ldots, v_k$ are job local variables initialized by the evaluations of the expressions $e_1, \ldots, e_k$. Their life-time is bound to the job and they may be accessed within f when captured by a top-level $bind$ expression.

2) Thread Identification: All threads within a group evaluate the call $f()$, hence they are processing the same sequential code. Furthermore, all threads in a group are indexed. Index dependent control flow can be used to cause execution traces to diverge. The functions

$$getThreadID : (uint\,\langle 4\rangle) \rightarrow uint\,\langle 4\rangle$$
$$getNumThreads : (uint\,\langle 4\rangle) \rightarrow uint\,\langle 4\rangle$$

are offered to retrieve the thread index and the thread group size from within a thread. The accepted parameter determines the nesting level. Passing 0 returns the index and size of the local group, passing 1 the corresponding values of the parent group and so forth.

3) Spawning and Merging: Every thread is allowed to create sub-thread groups using the function

$$spawn : (job) \rightarrow thread\_group$$

The resulting value of type $thread\_group$ references the newly started group. The function

$$merge : (thread\_group) \rightarrow unit$$

blocks until the referenced group has completed its job. In case several thread groups have been spawned, a call to

$$mergeAll : () \rightarrow unit$$

can be used to wait for the completion of all groups directly spawned by the processing thread.

4) Inter-Thread Communication: For a thread group to work cooperatively on a parallel job, means for communication are required. Three primitives are offered for this purpose – one enabling the distribution of work ($pfor$), one for redistributing data throughout a group ($redistribute$) and a third one for point-to-point communication ($channels$). The first two primitives are collective operators, hence in order for them to complete, all threads of a group must participate.

a) Work Distribution: Let $int = int\,\langle 4\rangle$. Work is distributed using the abstract operator

$$pfor : (int, int, int, (int, int, int) \Rightarrow unit) \rightarrow unit$$

which is named after its most prominent use case – the parallel for. The first three parameters define the range of an iterator (start, end and step size) while the last parameter accepts a function capable of processing sub-ranges of the given range. All threads within a group have to invoke this operator using the same set of arguments. The specified range will be distributed among the available threads using an undefined schema determined by the runtime environment. In case a specific scheduling policy should be enforced, it can be encoded directly within the IR. For instance, a $pfor$ can be replaced by a for-loop processing a share of the total range if a static scheduling should be realized. For dynamic loop scheduling approaches hints supporting the scheduling in the runtime system can be annotated [5].

After finished its shares, each thread will continue processing the following statement. There is no implicit barrier at the end. The only guarantee given is that after the last thread has completed the $pfor$ call, the entire range has been processed.

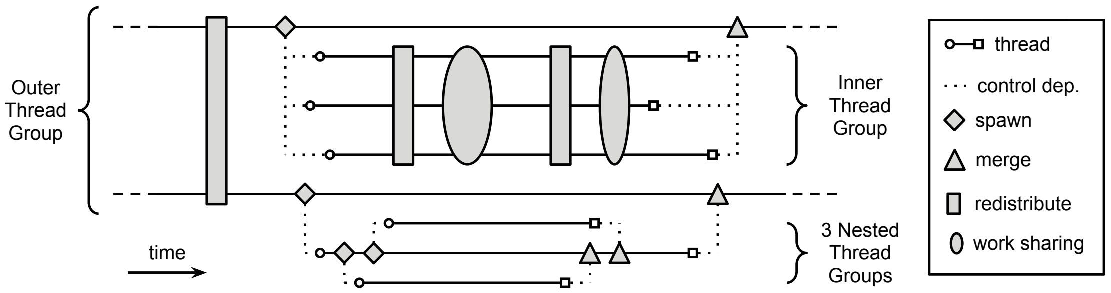

Fig. 2: Example execution snippet of a program based on INSPIRE's parallel model.

b) Data Distribution: Several parallel models provide primitives to scatter and gather data to and from all participating threads. These kinds of operation are particularly prominent in message passing solutions. As for the work-sharing, we aimed to identify a single primitive capable of covering all these functionalities. The result is the

$$redistribute : (\alpha, (array\langle\alpha\rangle) \Rightarrow \beta) \rightarrow \beta$$

operation. Similarly to $pfor$, this primitive is a collective operation and needs to be invoked by all threads within a group. However, unlike $pfor$ it is a blocking operation. Every thread contributes a piece of information via the first parameter of type α. The data of all threads is aggregated and passed as an argument to the function specified by the second parameter. The result of the evaluation of this function is returned as the result of the call to the $redistribute$ primitive. Thus, the operator simply collects all the contributions and allows a generic function to select the piece of information to be made available to the local thread.

Implementing collective operations based on an actual realization of the redistribute primitive would result in weakly performing applications. This internal representation is intended to unify collective operations for analyses, not to define their implementation. That is why derived constructs have been defined using this central primitive to encapsulate more specialized operations, including $barriers$ (in MPI, OpenMP and OpenCL), $reductions$ or $broadcasts$. As for most derived operations, their encoding is intercepted in the backend to generate specialized code.

c) Point-to-Point Communication: Finally, a mechanism to send information between individual threads is required. To solve this issue we borrowed the concept of *channels* encountered within model checking utilities [13]. We introduced the abstract generic type family $channel\langle\alpha, a\rangle$ and the operators

Operator	Type
$channel.send$	$(channel\langle\alpha, a\rangle, \alpha) \rightarrow unit$
$channel.recv$	$(channel\langle\alpha, a\rangle) \rightarrow \alpha$
$channel.probe$	$(channel\langle\alpha, a\rangle) \rightarrow bool$
$channel.create$	$(type\langle\alpha\rangle, param\langle a\rangle) \rightarrow channel\langle\alpha, a\rangle$
$channel.release$	$(channel\langle\alpha, a\rangle) \rightarrow unit$

The first type parameter α determines the type of data to be transferred trough a channel while the second a determines the buffer size of the channel. The $send$ operator inserts a new value into the buffer, while the $recv$ operator takes values from the buffer in order. Both may block the execution of the calling thread in case the buffer is full ($send$) or empty ($recv$).

Besides transferring data, channels are also used to model locks and bounded semaphores. For instance, a channel of the type $channel\langle unit, 1\rangle$ is equivalent to a binary semaphore when using the $channel.send$ as the V and $channel.recv$ as the P operator.

d) Communication using Shared Memory: INSPIRE does not prohibit the exchange of information using references to shared memory locations. However, as usual, the content of memory locations updated by other threads remains undefined until a synchronization between the source and target thread occurs via a common call to $redistribute$ or the forwarding of information using a $channel$.

A vital element used in high-performance applications to synchronize shared memory activities are atomic operations. Therefore we introduce a derived generic $atomic$ operator realizing a conditional, protected assignment to a single memory location. Specialized atomic operations can be built based on the generic operator. In the backend, derived atomic operations are mapped to corresponding instructions if available. Otherwise a locking mechanism is employed.

D. Overall Structure

Unlike other high-level IRs, INSPIRE does not strive to reflect the structure of translation units. Figure 3 illustrates how the C source depicted in (a) is represented. Within a typical AST based IR, a data structure representing a translation unit is created (b). The root node represents an organizational structure, e.g. a file, and lists a set of top-level definitions. Each definition is defined by sub-structures which may refer to top level elements or even external content.

INSPIRE follows a different approach. Instead of reflecting the organization of the input source files, it models the actual execution using a single expression. Figure 3 (c) illustrates the corresponding parse tree. The root element represents the entire execution of a code fragment, hence, typically the main function of a standalone application or some isolated library routine. Furthermore, whenever a function is called, the expression defining the target function is present right at the call site, independently from the translation unit it was originally defined in. This way, our IR provides a holistic view on the entire execution of a program, facilitating context sensitive or even whole-program optimizations.

The self-contained design of our IR allows for a variety of properties to be deduced from a given sub-tree without the requirement of any additional global context. Also, local modifications to a function f valid only within the current

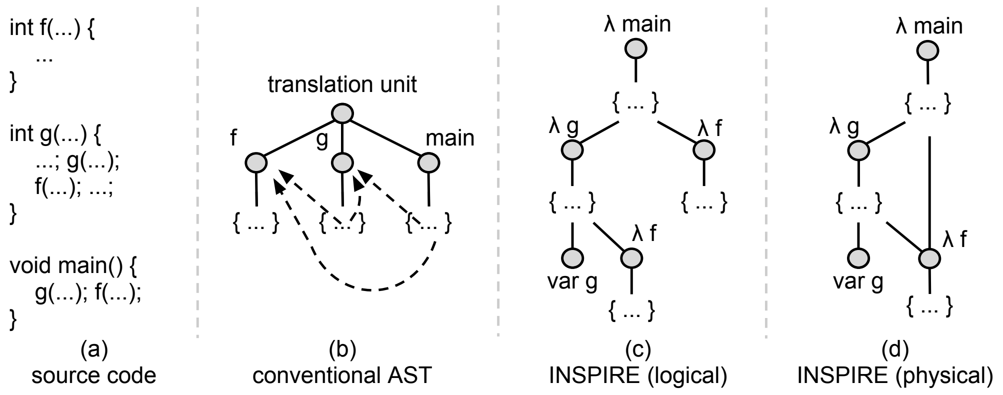

Fig. 3: Comparison of IR structures.

call-context (reflected by the path from the root node to the function) do not affect any other instance of f.

The downside of a representation following the schema of Fig. 3 (c) is the huge memory requirement due to the excessive duplication of functions and types. To alleviate this problem, nodes are shared in our implementation as illustrated in Fig. 3 (d). The logical tree structure is physically realized by a DAG. This enables the representation of IR codes consisting of several million logically addressable nodes (by their path from the root) using a few thousand physical nodes. For example, BT, the largest NAS benchmark [14] in terms of lines of code (2.281), is encoded in our IR using 10.296.189 addressable logical nodes, yet physically realized using only 12.549 nodes, resulting in a total memory consumption of 1.8 MB. A similar feature has been previously investigated for a limited set of IR node types [15], yet not utilized for the full IR.

To support the effective sharing of nodes in the IR DAG, all nodes have to be immutable. Consequently, whenever altering IR structures, new nodes linking existing IR sub-trees in a different manner are constructed. The effort of doing so is limited due to the locally restricted nature of this kind of tree modifications. Changes can therefore be conducted efficiently.

Impact on the Frontend: For the C/C++ frontend the overall structure of INSPIRE results in two major consequences. To begin with, all definitions of functions referenced by an input application have to be collected from potentially multiple translation units. Similar undertakings for realizing whole program representations have been proposed before [8], [15]. The second consequence results from the lack of global definitions within our IR. Every function within INSPIRE is self-contained, hence it exclusively refers to data obtained via its parameters. However, global variables are frequently encountered within scientific C codes and benchmarks. To compensate for this restriction, our frontend collects all global and static variables within the input code and aggregates them into a `struct`. This struct is initialized in the function forming the entry point of the encoded application and forwarded by reference to every function requiring access. Furthermore, in the backend, structs being allocated in the entry point function are re-globalized to avoid losing the performance benefit of global data structures.

E. Comparison with other IRs

Besides the novel unified representation of parallel control flows, INSPIRE has been designed to be minimal. Clang [10], for instance, utilizes ~ 150 different node types to represent C types, expression and statements, ~ 30 additional to cover C++ codes. ROSE [7] defines more than 330 node types. In both cases, node types for modeling the structure of translation units are not even included. Admittedly, those architectures have been designed to cover every syntactical detail of the original input code. Like CIL [8], which only requires ~ 40 node types for representing types, statements and expressions, our IR focus exclusively on relevant semantic aspects of input codes. In total, INSPIRE defines 7 type, 8 expression and 8 statement types. For the C++ support, 3 additional type constructs ($constructor$, $destructor$ and $member\ function$) and 2 extra statements ($throw$ and $try\ ..\ catch$) are needed. Clearly, a small, assessable set of node types simplifies the implementation of analyses, transformations and other utilities – in particular when defining functionality inductively over the structure of the IR.

IV. Examples

A. OpenMP

Encoding most OpenMP constructs using our IR is straight forward. The $parallel$ construct creating a parallel scope is converted into a call of the form

$$merge(spawn(X))$$

Since in OpenMP the thread reaching the parallel block is suspended, the $merge$ call directly encloses the $spawn$. The required job expression X is derived from the block statement annotated by the OpenMP $parallel$ pragma. For its extraction, data environment clauses like $private$, $shared$ and $firstprivate$ need to be respected by capturing references to variables of the surrounding context, creating local substitutes within the job body and initializing them accordingly.

The OpenMP work sharing constructs for, $single$, and $sections$ are encoded based on the $pfor$ primitive using the iterator range of the associated loop, a single iteration, or one iteration per $section$. An additional $barrier$ is appended in case no $nowait$ clause is provided. A barrier is equivalent to the call

$$redistribute\left(unit, \left(array\left\langle unit\right\rangle\ data\right)\left\{return\ unit;\right\}\right)$$

where $unit$ is the constant representing the one instance of the $unit$ type.

OpenMP tasks are modeled using IR jobs with a range limited to 1, resulting in the creation of a thread group consisting of a single thread. Since our parallel model allows for recursively nested parallelism, tasks may spawn additional tasks as required. The $taskwait$ primitive available for joining tasks in OpenMP is directly translated into a call to the $mergeAll$ primitive. Finally, means of synchronization including critical regions and locks are modeled using channels of the type $channel\,\langle unit, 1\rangle$.

B. Cilk

For implementing nested recursive parallel algorithms, Cilk, based on its two major keywords $spawn$ and $sync$, provides a more concise formalism than e.g. OpenMP does. Fortunately, both keywords can be easily modeled with our IR using the same constructs as for OpenMP tasks. The implicit $sync$ at the end of every function spawning tasks is also explicitly modeled using a $mergeAll$ call.

C. OpenCL

When running an OpenCL kernel, a function is being applied to every node within a 1, 2 or 3-dimensional grid in parallel. Thereby, the total grid, known as the *global range*, is partitioned into a large number of equally sized *local groups* of *work items*. Items within the same local group may share data using a fast shared memory segment. Also, synchronization means are exclusively offered within local groups.

This two-level decomposition and its various aspects need to be reflected within the IR since its proper utilization is the key for high-performance computing on accelerators. The two levels are encoded using two nested thread groups. The following code fragment illustrates the general schema:

```
merge(spawn(job[<global-range>]( <global-vars> ) {
  merge(spawn(job[<local-range>]( <local-vars> )
    { <kernel-code> }
}));
```

The outer level corresponds to the global range using one thread per local group and the inner level models the local groups. Within the kernel code, $getThreadID(0)$ can be used to obtain the index of the work item within the local group and $getThreadID(1)$ to access the group ID.

The rather extensive "boilerplate" code for device setup and kernel preparation surrounding an OpenCL kernel invocation in the host code is omitted by the frontend for clarity and re-added by the backend using APIs offered by our runtime environment. These also cover the required data transfers between memory segments.

The seamless integration of the kernel code into the host program enables the compiler to analyze and tweak the execution of the entire program (host and kernel part). For instance, successive kernel invocations processing the same data may be merged. Also, the data a kernel is applied to may be split and distributed among multiple devices [16].

D. MPI

Unlike for OpenMP, Cilk and OpenCL, in an MPI application the parallelism is neither confined to a single code region nor a single process. Fortunately, our parallel model is not restricted to threads within a single process. IR threads of a group can be distributed among multiple processes, as long as they do not share access to a common memory location – as it is always the case for MPI codes. Code fragments satisfying this restriction are accepted within the backend for generating distributed memory target code.

The whole-program parallelism of an MPI application can be modeled using a top-level $merge/spawn/job$ combination. Within the executed job, an array of communication channels of type $channel\,\langle msg, 1\rangle$ is created (COM-array). The utilized msg type is a struct modeling MPI messages and associated meta data, including their type, size and actual payload. The basic structure of an MPI program in the IR is the following

```
merge(spawn(job[1-inf] {
    array<channel<msg,1>> com =
        redistribute(channel.create(msg,1), id);
    /* ... MPI program body ... */
    channel.release(com[getThreadID(0)]);
  }));
```

where id is the identity function of type $(\alpha) \rightarrow \alpha$. This frame covers the obligatory MPI_Init and $MPI\_Finalize$ calls. However, unlike those it can be nested and processed multiple times. Since it is identical for all MPI applications, it has been isolated in a higher-order function $run\_distributed$ of type $(array\,\langle channel\,\langle msg, 1\rangle\rangle \Rightarrow unit) \rightarrow unit$ accepting the variable body part as a parameter.

Each channel within the COM-array is associated with one of the participating workers. If one MPI worker sends data to a peer using `MPI_Send` or similar primitives, the data is encapsulated into a message and submitted to the channel associated with the receiver using the $channel.send$ primitive. Consequently, MPI instructions receiving messages (e.g. `MPI_Recv`) are based on the $channel.recv$ operator accessing the channel associated to the local thread. All point-to-point MPI communication routines, including the asynchronous variants, can be modeled similarly.

Beside Send/Recv routines, MPI covers a rich set of collective operations. These kind of operations have been generalized by the IR's $redistribute$ operator which is specialized for the particular cases. For instance, an MPI reduction operation summing up the distributed values x and reporting the result exclusively to worker 0 can be modeled by

```
redistribute(x,
  (array<int<4>> data) {
    if (getThreadID(0) != 0) return 0;
    return sum(data);
  });
```

where sum is an ordinary sequential function summing up the elements of an array. Similar derived constructs can be used to represent *broadcast*, *scatter*, *gather* and *alltoall* routines.

Remark: Converting shared to distributed memory code is known to be a hard problem. Presenting instances of the problem in a different format can obviously not eliminate the underlying fundamental complexity. Within this paper we do not claim to provide any solutions for this problem. The goal of our IR is to offer one common infrastructure enabling the representation of both kind of applications to facilitate future research in this and other directions.

E. Other Languages

The parallel control-flow model of INSPIRE is generic enough to support a variety of additional parallel APIs, including *pthreads* and the C++11 threading facilities. Also, PGAS based approaches should be coverable by our IR. However, this issue has not yet been sufficiently investigated since no language based on this paradigm has been integrated into our system so far. Furthermore, our IR is not limited to C/C++. In particular our platform can serve as the foundation for implementing parallel domain specific languages (DSLs) which would benefit from the available infrastructure and from the implicit mapping of parallel constructs to sophisticated runtime system primitives within our backend.

V. Impact on Analyses and Transformations

The design of an IR can have a strong impact on the implementation of analyses and transformations built on top of it. This section briefly outlines some of those effects.

A. Analyses

The implementation of analyses is aided by several features of our IR. Its compact and unified set of formally specified (parallel) primitives eliminates the requirement of supporting a large variety of language-specific constructs. Also, the IR constructs themselves have been designed to avoid the need for excessive case distinctions. Furthermore, compared to low-level IRs, analyses can benefit from the preservation of "larger" expressions not being decomposed into low-level instructions, as has been carefully evaluated within [17]. The elimination of global and static variables removes the need for considering internal state of functions, and the holistic view provided by our IR facilitates whole-program analyses.

A very important tool when analyzing and manipulating application code these days is the polyhedral model [11]. As with other IRs, the matrix based description of static control parts (SCoPs) has to be extracted based on for-loops, conditions and array accesses. However, unlike within low-level or detailed high-level IRs these constructs are present within INSPIRE at a proper level of abstraction, simplifying the conversion in both directions.

On the downside, the functional nature of or IR requires the implementation of control flow analysis to deal with the *dynamic dispatch problem* – a problem that has to be dealt with anyway when analyzing object oriented languages exhibiting polymorphism.

The immutable, shared nature of the IR data structure and the capability of attaching annotations to nodes fosters the efficient implementation of analyses. The result of an analysis applied on an IR sub-tree can be annotated to its root node and reused the next time it is required. Since a node and all its child nodes are immutable, the result never has to be updated. Furthermore, since nodes are shared, the annotated results are reused within every context the corresponding sub-tree is encountered in, including future ones.

The Insieme compiler infrastructure currently offers polyhedral model based dependency analyses [6], [11] and runtime estimations [5], static and dynamic feature extraction utilities [18], support for the extraction and determination of the satisfiability of arithmetic constraints, and a framework for data flow analysis based on an inter-procedural parallel control-flow graph (I-pCFG). Some of those utilities are the subject of ongoing research.

B. Transformations

Utilities transforming IR codes mostly benefit from the self-contained tree structure, the holistic, execution oriented perspective, and the immutable, shared nodes of INSPIRE. The first property enables the creation and manipulation of IR fragments, independently from any global context. The second feature, together with our IR's functional characteristics, eliminates the boundaries between inter- and intra-procedural transformations. Further, the immutable and shared nodes provide a convenient way of preserving analysis information across transformations. Also, variants of code fragments can be efficiently constructed without the destruction of the original version or the conduction of a deep copy.

Within the Insieme infrastructure standard transformations including the inlining or outlining of functions, the specialization of functions by fixing parameters, constant propagation and collapsing, algebraic simplifications, (polyhedral) loop transformations, and unfolding of recursive functions are covered. Also, a prototype of a term-rewriting engine enabling the high-level description of transformations based on the IR's tree structure has been integrated.

VI. Performance Evaluation

Ultimately, our infrastructure is intended to be utilized for researching performance enhancing techniques. Thus, the system itself should not have a significant inherent negative impact on the performance of processed codes. Clearly, a thorough and complete investigation of the performance impact of an IR compared to alternatives is infeasible as it would require the implementation and tuning of all possible transformations and optimizations on top of all investigated representations. Also, the influence of individual design decisions and the implementation of the frontend or the backend can not be investigated without the effort of implementing a variety of alternatives.

Nevertheless, to provide a meaningful characterization of the performance impact of the conversion steps inherent in our environment we set up a test case which forwarded the result of the frontend directly to the backend without applying any optimizations. This corresponds to the basic setup researchers encounter when starting the investigation of a problem using our platform[5]. The resulting code should ideally exhibit the same performance as the original, unprocessed code. For our evaluation we used the NAS Parallel Benchmarks [14], the PolyBenchs/C [19], the Barcelona OpenMP Tasks Suite [20]

[5]sources may be acquired from the authors; distributed runtime support for MPI is still under development

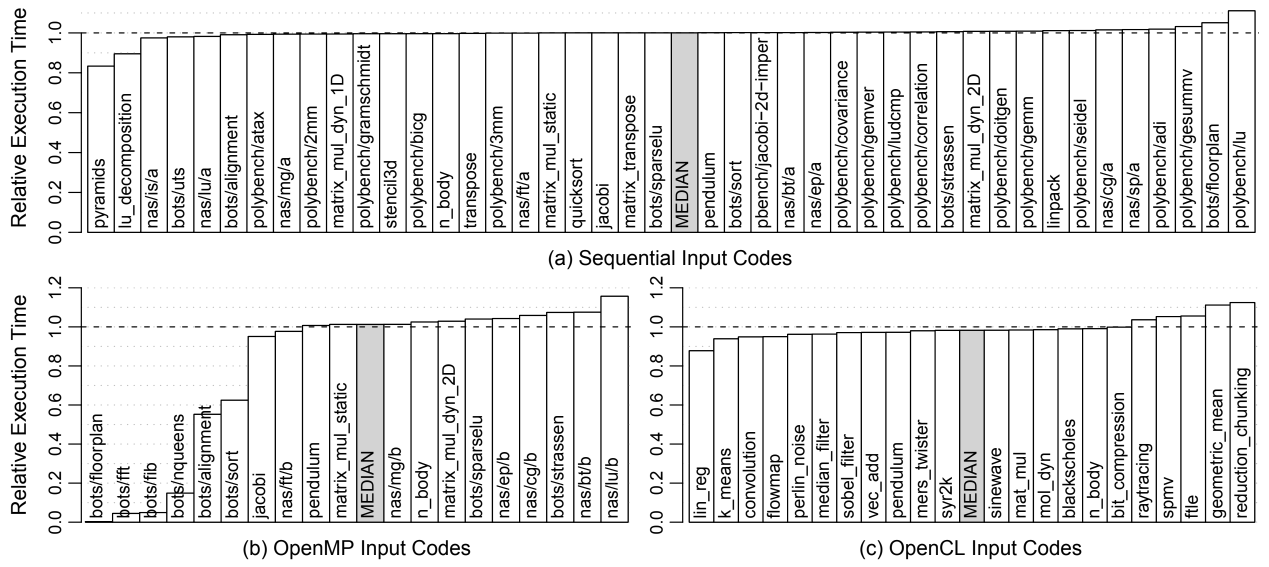

Fig. 4: Relative execution time ($t_{insieme}/t_{original}$) of benchmarks processed by a plain Insieme setup.

and a few additional implementations of algorithms frequently encountered within scientific applications. All codes have been compiled using GCC 4.6.3 with optimizations enabled (`-O3`). The relative execution time of the processed codes compared to the original sources is illustrated within Fig. 4.

As shown within (a), for the vast majority of codes, the intermediate step via our IR does not induce any significant effect on their sequential performance. The execution times of the OpenMP test cases (b) on the other hand is heavily influenced by the capabilities of our runtime system. It has been specifically designed to deal with task parallelism, resulting in a major improvement for corresponding applications. For codes utilizing loop-parallelism (∼right half), the fact that user-defined scheduling policies are not forwarded to the runtime in this experimental setup results in a slight slowdown since the default policy is less efficient. However, automated, compiler-supported loop scheduling based on our system has been demonstrated to result in significant performance improvements in these cases [5].

Also, OpenCL codes (c) benefit from the efficient combination of OpenCL related management operations within the runtime system, leading to a slight improvement for the majority of cases. However, for a few examples the combination of OpenCL kernel and device management operations re-added by the backend is not yet as efficient as the one used within the original implementation.

The applicability of our design and infrastructure for conducting research in the domain of parallel applications is further demonstrated by the list of derived work which has already been based on our platform [5], [6], [16], [18].

VII. Related Work

A large number of internal representations and languages for compiler infrastructures has been presented in the past. The solutions range from low-level, assembly-like IRs including GCC's RTL, the LLVM IR [21], and Java Bytecode [22], over mixed-level IRs such as GCC's GENERIC [23], SUIF2 [24], and SAIL [25] to high-level representations like the LLVM Clang AST [10], IPR [15], and SAGE III [7] utilized by ROSE. The latter is the preferred level for analysing and manipulating coarse-grained parallel codes. However, their close resemblance to high-level source languages makes the implementation of analyses and transformations a challenging task. This problem has been tackled by CIL [8] using a reduced variant of C to simplify analyses. None of the listed IRs covers parallel aspects within their specification.

Early attempts to add concurrent concepts to SSA based IRs focused on a restricted set of parallel codes based on *cobegin/coend* parallel sections [26], [27]. Pop et al. [28] presented a mechanism enhancing the support for full-blown, OpenMP like, parallel constructs within IRs designed for single-threaded codes. It preserves the optimization potential of classical low-level, sequential optimizations in the presence of parallel constructs while our approach introduces potential for coarse-grained, high-level program restructuring operations and parallel optimizations.

The Sequential to Parallel Intermediate Representation Extension (SPIRE) has been recently proposed to augment mid- to high-level IRs to cover parallel constructs [29]. The suggested ten primitives follow a similar theme as our own approach and can be simulated by INSPIRE constructs. However, SPIRE is lacking the concept of thread groups and hence the possibility of collective work and data sharing constructs. These constructs are required when encoding e.g. non-trivial OpenMP parallel loops contained within larger parallel regions as well as for structuring nested parallelism.

Erbium [30] defines an IR for streaming applications to handle fine-grained thread parallelism. It is at the same time an IR, a low-level programming language and a fast runtime implementation. While INSPIRE is a proposal for unifying high-level parallel APIs, Erbium provides an efficient solution for a specific domain of applications.

On a higher level of abstraction, parallel skeletons [31] offer higher-order functions to structure parallel codes. Implementations, including P3L [32] and the Delite IR [33] require the programmer to explicitly or implicitly map their algorithms to a set of predefined patterns. While skeletons are focusing on higher order patterns encountered within parallel applications, our work identifies a set of primitives to describe constructs encountered within existing languages. Consequently, skeletons can be specified on top of our IR.

Finally, research compilers converting OpenMP pragmas and other directives into library calls have been implemented based on C/C++ AST manipulations [34], [35]. None of those considers parallelism within their IR. The coordination and tuning of parallelism is left to the targeted runtime system.

VIII. Conclusion

Within this paper we presented a novel approach for a high-level compiler IR providing a common representation for a variety of parallel languages. By enforcing the utilization of a universal parallel model, the substantial differences in the syntactic and semantic description of parallelism within our source languages are eliminated. Furthermore, by supporting the parallel model on a language level, seamless integration with the rest of the representation has been achieved. Utilities based on our IR benefit from the unified representation, the elementary breakdown of rich end-user oriented constructs into a limited set of primitives modeling only essential functionality, and the holistic, execution-oriented, overall structure. Furthermore, its concise structure – resulting in a significantly reduced number of node types compared to other IRs – is sufficient to preserve performance relevant aspects of processed codes. A variety of derived work built upon our system demonstrates its practical applicability for tuning parallel C/C++ applications. We therefore conclude that INSPIRE and its supporting infrastructure provide a valuable platform for future research in the area of parallel languages and optimizing compilers.

Acknowledgment

The authors thank Hans Moritsch for numerous valuable discussions on the design of INSPIRE and coining its name.

References

[1] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes," in *PDP*, 2009.

[2] M. O'Boyle and F. Bodin, "Compiler reduction of synchronisation in shared virtual memory systems," in *ICS*, 1995, pp. 318–327.

[3] E. Petit, F. Bodin, and R. Dolbeau, "An hybrid data transfer optimization for gpu," *Compilers for Parallel Computers (CPC2007)*, 2007.

[4] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil, "Fast and efficient automatic memory management for gpus using compiler-assisted runtime coherence scheme," in *PACT*, 2012, pp. 33–42.

[5] P. Thoman, H. Jordan, S. Pellegrini, and T. Fahringer, "Automatic OpenMP loop scheduling: a combined compiler and runtime approach," *OpenMP in a Heterogeneous World*, pp. 88–101, 2012.

[6] S. Pellegrini, T. Hoefler, and T. Fahringer, "Exact dependence analysis for increased communication overlap," in *EuroMPI*, 2012, pp. 89–99.

[7] "ROSE user manual: A tool for building source-to-source translators," 2012. [Online]. Available: http://rosecompiler.org/ROSE_UserManual

[8] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "Cil: Intermediate language and tools for analysis and transformation of c programs," in *CC*, 2002, pp. 213–228.

[9] "Insieme compiler and runtime infrastructure." Institute of Computer Science, Distributed and Parallel Systems Group, University of Innsbruck. [Online]. Available: http://insieme-compiler.org

[10] "clang: a C language family frontend for LLVM," October 2012. [Online]. Available: http://clang.llvm.org

[11] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The polyhedral model is more widely applicable than you think," in *CC*, 2010, pp. 283–303.

[12] B. Pierce, *Types and programming languages*. MIT press, 2002.

[13] G. Holzmann, "The model checker spin," *Software Engineering, IEEE Transactions on*, vol. 23, no. 5, pp. 279–295, 1997.

[14] D. H. Bailey, E. Barszcz, J. T. Barton *et al.*, "The NAS parallel benchmarks," The International Journal of Supercomputer Applications, Tech. Rep., 1991.

[15] G. Dos Reis and B. Stroustrup, "A principled, complete, and efficient representation of C++," *Mathematics in Computer Science*, vol. 5, no. 3, pp. 335–356, 2011.

[16] I. Grasso, K. Kofler, B. Cosenza, and T. Fahringer, "Automatic problem size sensitive task partitioning on heterogeneous parallel systems," in *PPOPP*, 2013, pp. 281–282.

[17] F. Logozzo and M. Fähndrich, "On the relative completeness of bytecode analysis versus source code analysis," in *CC*, 2008, pp. 197–212.

[18] H. Jordan, P. Thoman, J. J. D. Barrionuevo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch, "A multi-objective auto-tuning framework for parallel codes," in *SC*, 2012, p. 10.

[19] L.-N. Pouchet, "PolyBench: The polyhedral benchmark suite." [Online]. Available: http://www.cse.ohio-state.edu/~pouchet/software/polybench

[20] A. Duran, X. Teruel *et al.*, "Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP," in *ICPP*, 2009, pp. 124–131.

[21] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004, pp. 75–86.

[22] T. Lindholm and F. Yellin, *Java virtual machine specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[23] J. Merrill, "Generic and gimple: A new tree representation for entire functions," in *GCC Developers Summit*, 2003, pp. 171–179.

[24] G. Aigner, A. Diwan *et al.*, "An overview of the SUIF2 compiler infrastructure," *Computer Systems Laboratory, Stanford University*, 2000.

[25] I. Dillig, T. Dillig, and A. Aiken, "SAIL: Static analysis intermediate language with a two-level representation," Stanford University Technical Report, Tech. Rep., 2009.

[26] J. Lee, S. P. Midkiff, and D. A. Padua, "Concurrent static single assignment form and constant propagation for explicitly parallel programs," in *LCPC*, 1997, pp. 114–130.

[27] D. Novillo, R. C. Unrau, and J. Schaeffer, "Concurrent ssa form in the presence of mutual exclusion," in *ICPP*, 1998, pp. 356–.

[28] A. Pop, A. Cohen *et al.*, "Preserving high-level semantics of parallel programming annotations through the compilation flow of optimizing compilers," in *CPC*, 2010.

[29] D. Khaldi, P. Jouvelot, C. Ancourt, and F. Irigoin, "Spire: A sequential to parallel intermediate representation extension," Technical Report CRI/A-487, MINES ParisTech, Tech. Rep., 2012.

[30] C. Miranda, A. Pop, P. Dumont, A. Cohen, and M. Duranton, "Erbium: a deterministic, concurrent intermediate representation to map data-flow tasks to scalable, persistent streaming processes," in *CASES*, 2010, pp. 11–20.

[31] M. I. Cole, *Algorithmic skeletons: structured management of parallel computation*. Pitman, 1989.

[32] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi, "P3l: A structured high-level parallel language, and its structured support," *Concurrency: Practice and Experience*, vol. 7, no. 3, pp. 225–255, 1995.

[33] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "A heterogeneous parallel framework for domain-specific languages," in *PACT*, 2011, pp. 89–100.

[34] C. Liao, D. J. Quinlan, T. Panas, and B. R. de Supinski, "A rose-based openmp 3.0 research compiler supporting multiple runtime libraries," in *IWOMP*, 2010, pp. 15–28.

[35] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta, "Nanos mercurium: a research compiler for openmp," in *Proceedings of the European Workshop on OpenMP*, vol. 8, 2004.