SEVENTH FRAMEWORK PROGRAMME
THEME
FET proactive 1: Concurrent Tera-Device Computing (ICT-2009.8.1)

PROJECT NUMBER: 249013

Exploiting dataflow parallelism in Teradevice Computing

D4.2 – Design of the unified intermediate representation

Due date of deliverable: 31/12/2010
Actual Submission: 31/12/2010

Start date of the project: January 1st, 2010 Duration: 48 months

Lead contractor for the deliverable: INRIA

Revision: See file name in document footer.

Project co-founded by the European Commission within the SEVENTH FRAMEWORK PROGRAMME (2007-2013)	
Dissemination Level: PU	
PU	Public
PP	Restricted to other programs participant (including the Commission Services)
RE	Restricted to a group specified by the consortium (including the Commission Services)
CO	Confidential, only for members of the consortium (including the Commission Services)

Change Control

Version#	Author	Organization	Change History
1.0	Albert Cohen	INRIA	
2.0	Albert Cohen	INRIA	Additional details
2.1	Albert Cohen	INRIA	Feedback

Release Approval

Name	Role	Date
Albert Cohen	Originator	
Albert Cohen	WP Leader	
Roberto Giorgi	Project Coordinator for formal deliverable	31.12.2010

TABLE OF CONTENTS

The following list of authors will be updated to reflect the list of contributors to the writing of the document.

Albert Cohen
INRIA

1. Glossary

OpenMP – Parallel programming pragma language on top of C, C++ and Fortran. In this deliverable, we refer to the OpenMP specification version 3.0 and to dataflow streaming extensions proposed in the TERAFLUX project (see D4.1).

http://www.openmp.org

HMPP – Hybrid Compiler for Manycore Applications, from CAPS entreprise, generally used as a shortcut for the HMPP pragma language on top of C, C++ and Fortran, and for the HMPP development workbench.

http://www.caps-entreprise.com

StarSs – StarSs is a task-based programming model that enables the exploitation of the applications' inherent parallelism at the task level. To mark the tasks in a StarSs application, annotations (pragmas) similar to the OpenMP ones are used. A uniqueness of StarSs tasks are the input, output or inout clauses that applied to tasks' parameters enable the runtime to track tasks' data dependences.

http://nanos.ac.upc.edu

Graphite – Graphite is the name of a R&D project and a compilation pass of the GNU Compiler Collection (GCC). It implements polyhedral compilation algorithms, applied to automatic parallelization and loop nest optimization. In this deliverable, the unified representation enables us to sketch an extension of Graphite to task-level optimizations and to enhance its analysis with static semantics carried by annotations of the efficiency languages.

Http://gcc.gnu.org/wiki/Graphite

2. Executive Summary

The traditional compilation methods for parallel languages consists in lowering the concurrency constructs to runtime library calls or hardware primitives in the early stages of the compilation flow. This approach is detrimental to performance, as it limits the scope of most compiler optimizations in presence of concurrency. It is also a waste of valuable static semantics for these compiler optimizations, as the parallel language constructs are not exploited in the static analyses of the optimization passes. It complicates the support for multiple alternative language syntaxes and semantics in a common backend compiler, although these syntaxes and semantics may share a lot of common principles and constructs. In this deliverable, we present the early design of a unified intermediate representation addressing all of these issues. In addition, our approach is designed to seamlessly integrate into an existing compiler framework with minimal impact to the optimization passes. An early implementation has been conducted in GCC, and will be used as a basis for future OpenMP and OpenMP dataflow extension support in the TERAFLUX project.

3. Introduction

This deliverable reports on the design a unified intermediate representation, aiming for the following properties: it should be compatible with different efficiency layer languages used in TERAFLUX, it should enable aggressive task-level optimizations, to adapt the grain of parallelism to the target, it should enable state-of-the-art loop optimizations, and finally it should enable classical compiler optimizations in presence of dataflow concurrency, as well as specific specialization steps to reduce the overhead of parallel and distributed execution.

3.1. Document structure

This deliverable contains one main technical section presenting the motivations for and the design of the unified representation.

3.2. Relation to other deliverables

This deliverable covers the compiler internals associated with the programming language designs discussed in D3.2, and with the techniques and tools described in D4.1 and D4.3.

3.3. Activities referred by this deliverable

This deliverable is associated with Task 4.1. The work on unified intermediate representations will extend into Tasks 4.2 for aggressive task-level optimizations and Task 4.3 for the optimizations dedicated to transactional memory.

4. Preserving the Semantics of Parallel Languages

We designed a unified representation of the high-level semantics of parallel programming languages in the intermediate representation of optimizing compilers. In general, the semantics of these languages does not fit well in the intermediate representation of classical optimizing compilers, designed for single-threaded applications, and is usually lowered to threaded code with opaque concurrency bindings through source-to-source compilation or a front-end compiler pass. The semantical properties of the high-level parallel language are obfuscated at a very early stage of the compilation flow. This is detrimental to the effectiveness of downstream optimizations.

We define the properties we introduce in our new representation and prove that they are preserved by existing optimization passes. We characterize the optimizations that are enabled or interfere with this representation and evaluate the impact of the serial optimizations enabled by this technique for concurrent programs, using a prototype implemented in a branch of GCC 4.6.

We wish to target all efficiency layer languages considered in TERAFLUX, including StarSs, HMPP and OpenMP streaming extensions. We would like to map these languages to the same, unified representation, benefiting from the same task-level optimizations and code generation algorithms targetting the TERAFLUX ISA.

This representation will be refined and fully implemented in the second and third years of the project, and will be the cornerstone of the optimizations taking place in Tasks 4.2 and 4.3.

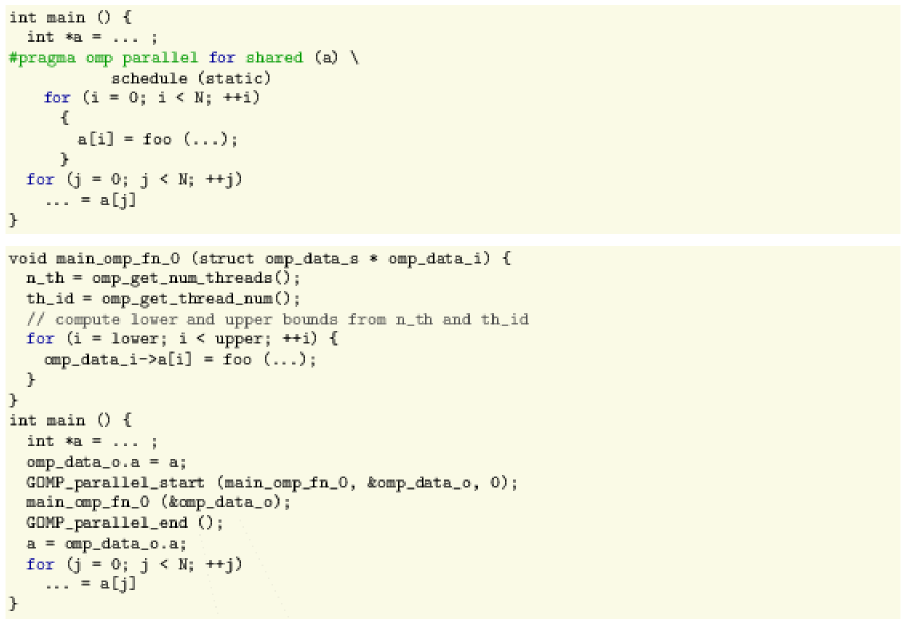
4.1 Motivation

The efficiency programming layer in the TERAFLUX project takes the form of annotation languages built on top of widespread, imperative languages. In our applications and benchmarks, we will focus on C and on the pragma-based annotations of the StarSs, OpenMP and HMPP languages. Notice these pragmas are written by programmers with moderate to high experience in parallel programming, while the Scala programming language has been selected in the project to represent productivity development practices and to offer a syntax and semantics for a wider range of programmers.

The early expansion of user annotations to runtime calls, with the associated code transformations, outlining, opaque marshaling of data and use of function pointers, is a process whereby concurrency is gained, at an early compilation stage, at the cost of the loss of the initial high-level information and obfuscation of the underlying code.

The annotations provide a wealth of precise information about data dependences, control flow, data sharing and synchronization requirements, that can enable more optimizations than just the originally intended parallelization.

The common approach for the compilation of parallel programming annotations is to directly translate them into calls to the runtime system at a very early stage. For example, in the GCC compiler, this happens right after parsing the source code. This means that all the high-level information provided by the programmer is lost and the compiler will have to cope with the resulting code obfuscation and loss of precise information. Our approach is to further abstract the semantics of the user annotations and bring this information into the compiler's intermediate representation using the technique presented in the next section. The semantical information is preserved, and when possible used or even refined, until the end of the code optimization passes, where it is finally translated to the intended runtime calls in a late expansion pass.



```
int main () {
  int *a = ... ;
#pragma omp parallel for shared (a) \
            schedule (static)
    for (i = 0; i < N; ++i)
      {
        a[i] = foo (...);
      }
  for (j = 0; j < N; ++j)
    ... = a[j]
}
```

```
void main_omp_fn_0 (struct omp_data_s * omp_data_i) {
  n_th = omp_get_num_threads();
  th_id = omp_get_thread_num();
  // compute lower and upper bounds from n_th and th_id
  for (i = lower; i < upper; ++i) {
    omp_data_i->a[i] = foo (...);
  }
}
int main () {
  int *a = ... ;
  omp_data_o.a = a;
  GOMP_parallel_start (main_omp_fn_0, &omp_data_o, 0);
  main_omp_fn_0 (&omp_data_o);
  GOMP_parallel_end ();
  a = omp_data_o.a;
  for (j = 0; j < N; ++j)
    ... = a[j]
}
```

Figure 1: The early expansion of a simple OpenMP example (top) results in information loss and obfuscation (bottom).

Let us consider the example on Figure 1 where a simple *omp parallel for* loop with a static schedule is expanded. Despite the fact that we chose one of the least disruptive expansions, the resulting code does not look quite as appealing for most analysis and optimization passes. If the original loop could have been unrolled or vectorized, it is now very unlikely it would still be.

To make matters worse, the resulting code is not only harder to analyze and optimize, but it also lost the information provided by the user through the annotations and we lost the capability of optimizing the parallelization itself. In the original version, as the loop is declared to be parallel with a shared data structure a, we know that the right-hand-side of the assignment $a[i] = \ldots$ is not partaking in any loop-carried dependences or that calls to the function foo have no ordering restrictions and can happen concurrently. In the expanded version, however, that information is lost and must be found through analyses that may, and quite likely will fail. Among other possibilities, the loop annotated as parallel may have been fused with the second loop, but that is no longer an option once expansion has taken place.

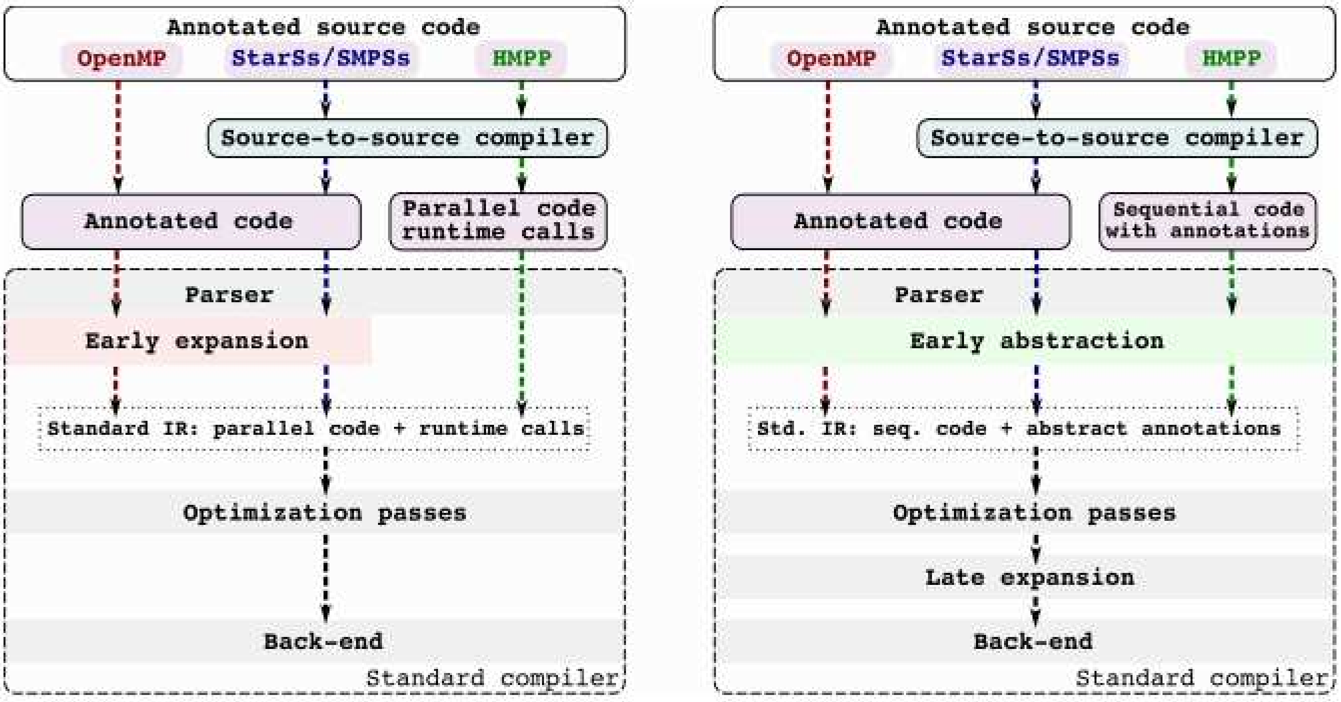


Figure 2: Compilation flow of high-level parallel-programming languages, current situation (left) and our objective (right).

Figure 2 illustrates the compilation flow of three parallel programming languages that are representative of this type of languages. OpenMP [9], StarSs [6] and HMPP [8] each in their own way suffer from this issue. StarSs and HMPP rely on source-to-source compilers as a first step. The source-to-source compiler they rely on is capable of generating optimized parallel code, either directly expanded to calls to the runtime system or translated into another high-level parallel programming language like OpenMP. From that point on, their compilation flow either goes through an early expansion pass that generates parallelized code and issues calls to the runtime along with OpenMP, or as is the case for HMPP, the code is parsed and directly represented in the compiler's intermediate representation. At that point, most of the potential for further optimization is lost.

In order to preserve the high-level semantics of user annotations and to avoid clobbering important optimizations or analyses, we replace the early expansion of user annotations by an early abstraction pass. This pass extracts the semantics of the annotations and inserts it into the compiler's original intermediate representation, using constructs that preserve the information in a state that is usable by analysis and optimization passes and that can ultimately be expanded to parallel code and runtime calls at the end of the compilation flow.

We believe that even languages like HMPP, with a dedicated optimizing compiler, can benefit from our approach as the source-to-source compiler is generally intended and specialized to perform the domain-specific optimizations corresponding to the original source language. Extending our framework to such a language should not be overly complicated, but to leverage the HMPP-specific optimizations, CAPS would need to write a new code generation backend for their source-to-source compiler targeting the unfied intermediate representation.

We attempt to address the following issues:

1. High-level parallel programming languages, in particular OpenMP, are poorly optimized by current compilers, even for simple and crucial sequential scalar optimizations.
2. Opportunities for optimizing the exploitation of parallelism are lost (e.g., possibility to compute optimized static schedules, verification ...).
3. User information on concurrency, dataflow and synchronization requirements is wasted. It can be used for more than only parallelization.

4.2. Semantic abstraction

The semantics of user-level annotations is generally defined with a direct correspondence to specific parallelization techniques or to specific runtime calls. Because of this, if instead of the early expansion we only represent the annotations, as they are, in the intermediate representation, the interpretation of their semantics will be necessary for each compiler pass that needs to use the information they carry. Multiple interpretation layers, in optimization passes and then in the late expansion pass, would severely reduce the genericity of this framework and make its extension cumbersome.

The solution we advocate is to replace the early expansion pass by an *early abstraction* pass that extracts the necessary information from user annotations and represents it using a unique set of abstract annotations irrespectively of the original language, lowering the annotations to a language-independent representation, which should provide a unified view of the user information whether it comes from OpenMP, HMPP or StarSs annotations.

The key insight is that the high-level user annotations mostly provide information on data-flow, with also some restrictions on control-flow that stem from the lack of precision on the dynamic data-flow. The concurrency is just a result of the absence of conflicts. We also recognize the importance of the additional information a user provides as hints on the best strategy, like for example which is the scheduling technique likely to yield the best results.

Adapting a new language, or an extension, to this early abstraction pass requires understanding and abstracting the underlying semantics of the annotations, but it should not require any modification in the optimization passes of the compiler. Additional target-specific semantics for new architectures or accelerators can easily be added in the form of user hints.

Following is the set of required abstract annotations, and a gist of their semantics.

Data-flow annotations.

- **use**: the variable or memory area is read within the associated block.
- **def**: the variable or memory area is written.
- **may-use**: the variable or memory area may be read within the associated block.
- **may-def**: the variable or memory area may be written.
- **safe-ref**: the variable is used or defined, but the user guarantees that all potential conflicts are handled, e.g., with manual synchronization.
- **reduction**: the associated variable is part of a reduction.

Control-flow annotations.

- **SESE**: the associated block of code is a Single-Entry Single-Exit region. There is no branching in or out and exceptions are caught within the region.
- **single**: the associated block can only be executed on one thread.
- **point-to-point synchronization**: typically a producer-consumer dependence.
- **barrier**: collective synchronization, either an explicit barrier or when a barrier is implied at the end of a block.
- **transaction**: atomic and isolated control flow region.
- **memory barrier**: a memory flush is required at this point.

These annotations can be further qualified with the precise memory model (e.g., consistency on synchronization points) and any additional language-specific properties.

User hints.

Many of the decisions involved in tuning the parallel code generation and the execution are hard to decide from static analysis alone. We store as *hints* all the information provided by the programmer. If the optimization passes can find provably better choices, then these hints can be ignored, otherwise they should take precedence.

- **parallel**: this hint may be important for loops, because even if static analysis can recognize the loop is parallel, the profitability of the parallel execution may not be obvious. If the programmer annotates a loop as parallel, it should not be overlooked.
- **schedule**: the choice of the schedule for a parallel loop.
- **num_threads**: number of threads available.
- More generally, any language-specific or target-specific information can be stored as a hint. In particular, in case the late expansion pass is too difficult to perform using the abstract annotations alone, it would be trivial to keep the whole set of original annotations in this form. As we will see in Section 4.4, this is the easy way to solve the problem of enabling classical sequential optimizations for such languages as OpenMP.

These abstract annotations provide readily usable information to the optimization passes. They can also be refined through static analysis as, for example, OpenMP sharing clauses will generally only provide *may-def/may-use* information which can be promoted to *def/use*.

Depending on the compiler pass, annotated blocks of code can be either seen as black boxes, that have well-specified memory effects and behaviour, or they may need to be perfect white boxes to allow unrelated optimizations to be transparently applied. The representation of these annotations needs to allow access to the code, yet restrict optimizations that would break the semantics of the optimizations.

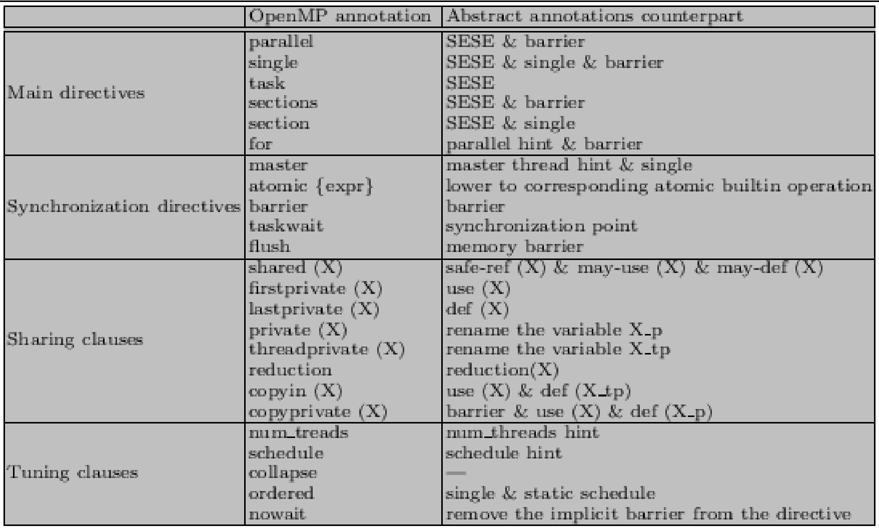
Default clauses.

In languages that have default clauses, or default specified behaviour, all defaults must be made explicit by the early expansion. This is part of the interpretation of the language's semantics and keeping any information implicit would hamper the genericity of the approach. The abstract annotations should be self-contained.

In particular, the OpenMP default sharing or a *default* clause allows the programmer to leave some of the sharing clauses implicit. We convert all implicit clauses to explicit ones during the early abstraction pass, which allows to decouple the intermediate representation from the OpenMP-specific semantics of the default sharing.

Example: abstract semantics for OpenMP.

Without attempting to provide a full characterization of the OpenMP semantics, we present on Figure 3 a subset of the abstract semantics of the language.



	OpenMP annotation	Abstract annotations counterpart
Main directives	parallel	SESE & barrier
	single	SESE & single & barrier
	task	SESE
	sections	SESE & barrier
	section	SESE & single
	for	parallel hint & barrier
Synchronization directives	master	master thread hint & single
	atomic {expr}	lower to corresponding atomic builtin operation
	barrier	barrier
	taskwait	synchronization point
	flush	memory barrier
Sharing clauses	shared (X)	safe-ref (X) & may-use (X) & may-def (X)
	firstprivate (X)	use (X)
	lastprivate (X)	def (X)
	private (X)	rename the variable X_p
	threadprivate (X)	rename the variable X_tp
	reduction	reduction(X)
	copyin (X)	use (X) & def (X_tp)
	copyprivate (X)	barrier & use (X) & def (X_p)
Tuning clauses	num_treads	num_threads hint
	schedule	schedule hint
	collapse	—
	ordered	single & static schedule
	nowait	remove the implicit barrier from the directive

Figure 3: OpenMP semantics.

Adapting this framework for an OpenMP extension for streaming [4,7], consisting in two additional clauses for task constructs, would require also adding the same two data-flow annotations. This extension defines an input and an output clauses for tasks, which can be abstracted to a `use` and a `def` annotations in the simple, scalar version of the extension.

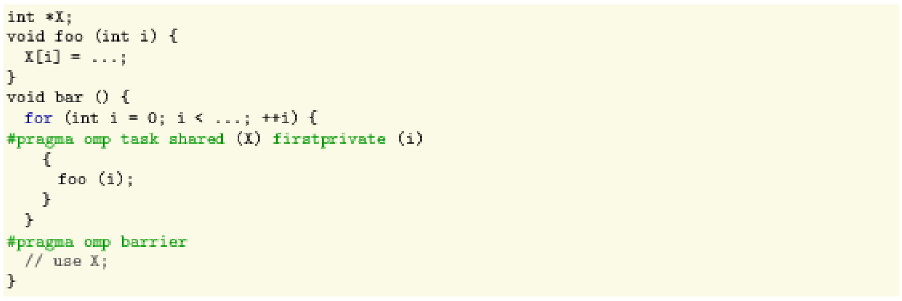
4.3. Intermediate representation

In this section we present a simple yet convenient way to represent high-level information in the current intermediate representation of optimizing compilers, in a way that does not require special care.

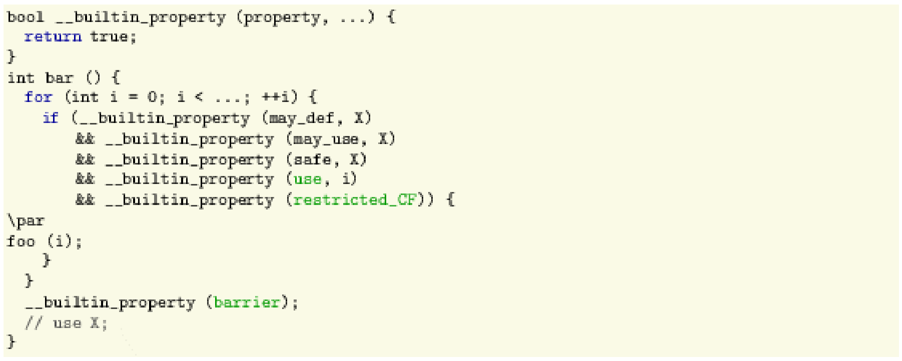
The semantics of user-level annotations is generally defined with a direct correspondence to specific parallelization techniques or to specific runtime calls. This makes them well-suited for early expansion as they are self-contained and require no static analysis or verification. A direct translation, or expansion, can be performed at the earliest stages of the compilation flow, which is a convenient way to avoid the interactions with the optimization passes of compilers.

A common constraint in extending the intermediate representation of a compiler is that it requires modifying most compiler passes, if only to keep the new information consistent after code transformations. Instead of modify the representation, we circumvent this issue by making use of the existing infrastructure. We introduce calls to factitious builtin functions and conditional statements that allow us to carry the abstract semantics of the user annotations and also to prevent aggressive optimizations that would break the parallel semantics intended by the user.

As Figure 4 shows, we use variadic builtins, with parameters corresponding to the abstract annotation properties and, when relevant, the program variables to which the property applies.



```
int *X;
void foo (int i) {
  X[i] = ...;
}
void bar () {
  for (int i = 0; i < ...; ++i) {
#pragma omp task shared (X) firstprivate (i)
    {
      foo (i);
    }
  }
#pragma omp barrier
  // use X;
}
```



```
bool __builtin_property (property, ...) {
  return true;
}
int bar () {
  for (int i = 0; i < ...; ++i) {
    if (__builtin_property (may_def, X)
        && __builtin_property (may_use, X)
        && __builtin_property (safe, X)
        && __builtin_property (use, i)
        && __builtin_property (restricted_CF)) {
\par
foo (i);
    }
  }
  __builtin_property (barrier);
  // use X;
}
```

Figure 4: Builtins.

It would be quite easy at this point to not perform any abstraction and only focus on avoiding the code obfuscation of the early expansion, by simply representing directly all of the language's annotations and performing a late expansion after the sequential optimization passes. This is, however, only a partial and suboptimal result.

One of the imperative requirements to make our representation robust, despite not requiring to modify optimization passes, is that it naturally prevents any transformation that would invalidate the semantics of the annotations.

Many compiler passes have the potential to break the semantics if they are to perform without any constraint. However, the representation implicitly introduces a few constraints that we believe to be sufficient. The conditional expressions it introduces, relying on opaque builtin function calls, ensure the integrity of the blocks of code they are attached to.

4.4. Application to compiler analysis and optimization

The information provided by programmers through high-level annotations has the potential to be of great use in other areas of compiler analysis and optimization than only parallelization. The first major benefit of our technique is that it allows to avoid the systematical loss of classical sequential compiler optimizations when compiling parallel programming languages. In a second time, we survey some other areas where we have hope to make an impact using the information gathered from the programmer annotations.

We have already started to experiment with using this information for extending the code coverage of the Graphite polyhedral optimization framework and we believe it will prove very useful for improving the accuracy of some analysis passes, like for example data-dependence and pointer alias analyses. Finally, a more productivity-oriented advantage of this scheme, we will discuss the potential for compiler verification of the program annotations.

4.5. Code obfuscation and optimization inhibition

One of the main drawbacks of the early expansion pass is that it leaves little room for classical sequential optimizations, some of which have much potential for improving performance. Optimizing concurrent applications is made harder by the presence of parallelization code. By postponing the expansion pass, we allow the compiler to apply these optimizations before generating the parallelization code, as long as we can ensure that the semantics are preserved.

A sequential optimization pass will, in most cases, not interact with our representation and will therefore consider any annotated block of code as a white box. For example, on Figure 5, the optimization pass will consider that the conditional statement and the call to our builtin function is simply user code. In order to ensure that the compiler can efficiently analyze white boxes, the builtin function is typed so that the access to a variable from the builtin function matches its semantics. More specifically, on Figure 5, the builtin function has const parameters. This means that this code can easily be analyzed to show that it only reads x, thus enabling for example a constant propagation pass. The invalidation of this *property*, in case the x variable is substituted by a constant value in both references, does not invalidate the semantics of the annotations.

```
if (__builtin_property (use, x)) {
  ... = ... x ...;
}
```

Figure 5: As a white box, the builtin function is considered as user code.

It appears clear that the constant propagation would not be possible if, for example, the assignment statement on Figure 5 was enclosed within an OpenMP task construct that had been expanded. In such a case, the value of x would have been marshalled in an opaque data structure and passed to a function pointer in the same way as the expansion presented on Figure 1.

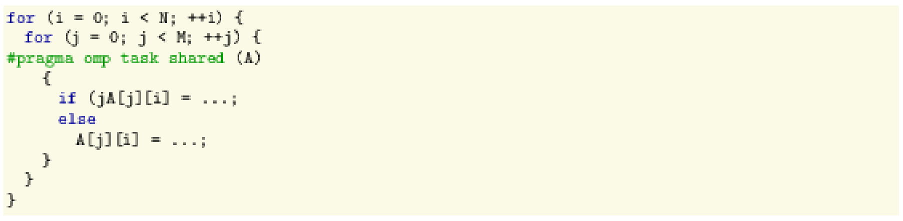
Despite major efforts, data-parallel and transactional extension of imperative languages still incur significant overheads due to missed optimizations [10,1]. Our experiments demonstrate that optimization of parallel code can increase performance by up to $1.54\times$ on a real application, `FMradio`, thanks to vectorization and additional scalar optimizations alone [5].

4.6. Extending the scope of polyhedral optimization frameworks

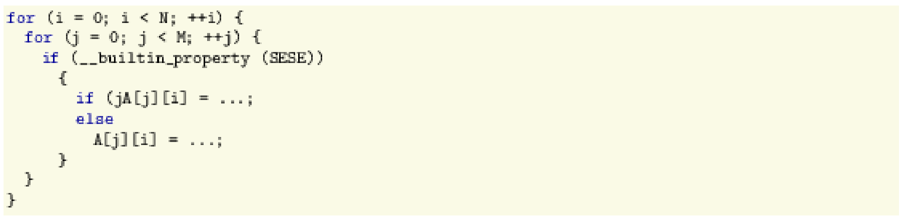
One of the traditional limitations of the polyhedral model has been its restriction to the representation and transformation of Static Control Parts (SCoPs) of programs. This restriction means that only static control is allowed and all array accesses must be through affine subscripts. This strong limitation reduces its applicability. Recently, Benabderrahmane et al. proposed a simple extension of the code generation algorithm and a generic scheme to capture dynamic, data-dependence conditions in polyhedral compilation frameworks [3]. This approach can represent arbitrary intraprocedural, structured control flow. Yet it is only a conservative approach, where dependences remain computed through static analysis, and where complex control flow or irregular data structures (with pointers) may result in rough approximations [2]. In addition, it is only an intraprocedural extension.

We advocate for a complementary approach, using annotations to drive the formation of larger SCoPs. While maintaining the static control properties, this approach allows for more accurate dependence analysis and enables more aggressive optimizations. We modified GCC's polyhedral optimization framework, Graphite, to use the abstract annotations in the SCoP detection phase. By assimilating well-behaved blocks of code (corresponding to the SESE abstract annotation), with the proper memory effects information, to black boxes that are represented as single statements in the polyhedral model, we hide non-static control flow or non-affine array subscripts from the optimization framework without compromising its correctness.

Let us consider the example on Figure 6 of non-static control code that is currently, and correctly, not recognised as a SCoP by Graphite and thus not optimized.



```
for (i = 0; i < N; ++i) {
  for (j = 0; j < M; ++j) {
#pragma omp task shared (A)
    {
      if (jA[j][i] = ...;
      else
        A[j][i] = ...;
    }
  }
}
```



```
for (i = 0; i < N; ++i) {
  for (j = 0; j < M; ++j) {
    if (__builtin_property (SESE))
      {
        if (jA[j][i] = ...;
        else
          A[j][i] = ...;
      }
  }
}
```

Figure 6: Extending Static Control Parts.

If the task directive is expanded early within the existing OpenMP framework, this would be a lost cause for Graphite as there would be opaque function calls and marshaling of a pointer to the array in an opaque data structure. If the task directive is ignored, then the non-affine modulo conditional expression makes the SCoP detection fail.

However, using our representation and considering the task as a black box within Graphite enables the optimization of this loop nest. The current implementation of the early abstraction pass is already handling common OpenMP constructs. The Graphite adaptation to represent single-entry single-exit regions as black boxes and to use the information we extracted from the OpenMP annotations is complete and will be included in the next release of GCC.

We plan to test the benefits of this technique by compiling OpenMP benchmarks in this way and compare to the sequential execution of the programs. As the late expansion pass from our representation to generate parallel code is still under development.

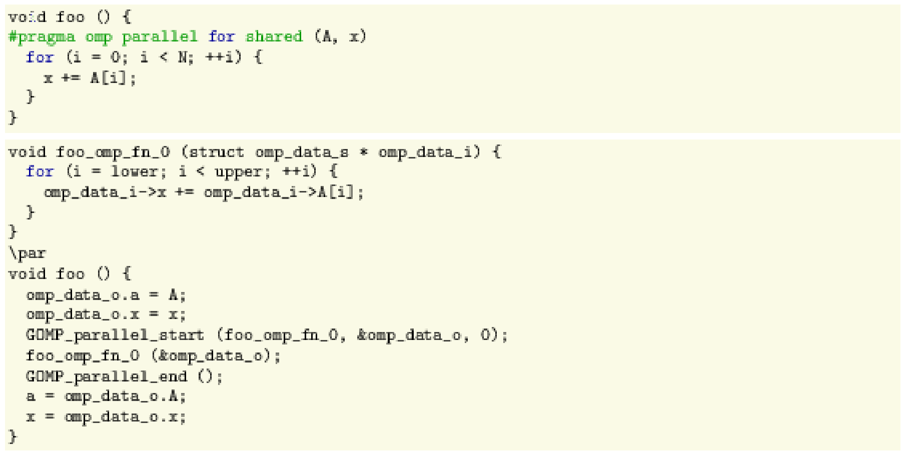
Combining this annotation-based SCoP formation method with Benabderrahmane's extension [3] is an exciting future work. It will motivate additional support from annotations to refine the quality of the data dependence and pointer aliasing computation.

4.7. Statically verifying user annotations

For languages like OpenMP, where the early expansion only consists in a direct translation of the directives to parallelization code, the compiler can only perform rudimentary sanity checks along the line of verifying that the same variable does not appear on more than one sharing clause. This is a serious limitation to productivity as most mistakes must be tracked through debugging.

Performing the expansion at a late stage will ensure the compiler has gathered much more information on the program through static analysis and will be able to more accurately and more completely assess the validity of the user annotations.

For instance, relying on user annotations does not mean that static analysis can be forgotten. It is important to compare its results with the programmer information. If there is a contradiction and the static analysis gives a precise answer, then there is a reasonable case for considering the programmer made a mistake.



```
void foo () {
#pragma omp parallel for shared (A, x)
  for (i = 0; i < N; ++i) {
    x += A[i];
  }
}
```

```
void foo_omp_fn_0 (struct omp_data_s * omp_data_i) {
  for (i = lower; i < upper; ++i) {
    omp_data_i->x += omp_data_i->A[i];
  }
}
\par
void foo () {
  omp_data_o.a = A;
  omp_data_o.x = x;
  GOMP_parallel_start (foo_omp_fn_0, &omp_data_o, 0);
  foo_omp_fn_0 (&omp_data_o);
  GOMP_parallel_end ();
  a = omp_data_o.A;
  x = omp_data_o.x;
}
```

Figure 7: The wrong code annotation, missing the reduction clause on x (top) and the result of early expansion (bottom).

Let us consider the example presented on Figure 7, where the programmer omitted a reduction clause. The code is obviously incorrect. If the annotations are expanded early, even though it is possible for the compiler to detect, at a later stage, the reduction in the function $foo\_omp\_fn\_0$, there is no information left about the original annotation, not even about the fact that this is a parallelized loop.

If the early abstraction pass was used instead, as soon as the compiler detects the dependence on x, or the reduction, it is possible to decide that the programmer made a mistake as he declared the loop to be parallel.

4.8. Roadmap for future work

In order to experimentally validate our approach and evaluate the impact these techniques have on real applications, we envisage the following roadmap:

- Evaluate the additional code coverage that can be achieved in the polyhedral representation by using the additional semantics of OpenMP annotations in the programs of the OpenMP Benchmark Suite.
- Consider streaming OpenMP dataflow streaming extensions carrying explicit dependence information, to enhance the accuracy of data dependence analyses.
- Further evaluate the performance improvement this added coverage has on both the late-expanded version and on the sequential version.
- Evaluate more precisely and more extensively the impact of missed optimization opportunities on the OpenMP Benchmark Suite, by comparing the performance achieved using the original OpenMP code with the classical early expansion to the performance achieved using late expansion.
- Compare the performance results of early expansion to the results of both unoptimized late expansion and optimized late expansion with specific concurrency optimization.

5. Conclusions

We presented an alternative approach to the classical compilation flow of high-level annotation-based parallel programming languages. This alternative solution enables sequential optimizations of parallel codes, in particular it allows TERAFLUX efficiency layer programs to benefit from many optimizations that until now were out of reach. Further uses of the intermediate representation include the extension of the scope of polyhedral representation and optimization as well as static verification of user annotations. Eventually we will be able to use this intermediate representtion to map different TERAFLUX pragma-based languages to a common back-end targetting the TERAFLUX ISA.

6. References

1. W. Baek, C. C. Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun.
The opentm transactional application programming interface.
In *IEEE Intl. Conf. Parallel Architecture and Compilation Techniques (PACT'07)*, pages 376-387, 2007.
2. D. Barthou, J.-F. Collard, and P. Feautrier.
Fuzzy array dataflow analysis.
J. on Parallel and Distributed Computing, 40:210-226, 1997.
3. M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul.
The polyhedral model is more widely applicable than you think.
In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'10)*, number 6011 in LNCS, Paphos, Cyprus, Mar. 2010. Springer Verlag.
4. P. M. Carpenter, D. Ródenas, X. Martorell, A. Ramírez, and E. Ayguadé.
A streaming machine description and programming model.
In *SAMOS*, pages 107-116, 2007.
5. C. Miranda, P. Dumont, A. Cohen, M. Duranton, and A. Pop. Erbium: A deterministic, concurrent intermediate representation to map data-flow tasks to scalable, persistent streaming processes. In *Intl. Conf. on Compilers Architectures and Synthesis for Embedded Systems (CASES'10)*, October 2010.
6. J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta.
Hierarchical Task-Based Programming With StarSs.
Int. J. High Perform. Comput. Appl., 23(3):284-299, 2009.
7. A. Pop and A. Cohen. A stream-comptuting extension to OpenMP. In *Intl. Conf. on High Performance Embedded Architectures and Compilers (HiPEAC'11)*, January 2011.
8. S. B. R. Dolbeau and F. Bodin.
Hmpp: A hybrid multi-core parallel programming environment.
In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.
9. The OpenMP Architecture Review Board.
OpenMP Application Program Interface.
http://www.openmp.org/mp-documents/spec30.pdf.
10. C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai.
Code generation and optimization for transactional memory constructs in an unmanaged language.
In *ACM/IEEE Intl. Symp. on Code Generation and Optimization (CGO'07)*, pages 34-48, 2007.