Split Compilation: an Application to Just-in-Time Vectorization

Piotr Lesnicki Albert Cohen
Grigori Fursin
INRIA Futurs and LRI, Paris-Sud 11 University
fistname.lastname@inria.fr

Marco Cornero Andrea Ornstein
Erven Rohou
STMicroelectronics
firstname.lastname@st.com

Abstract

In a world of ubiquitous, heterogeneous parallelism, achieving portable performance is a challenge. It requires finely tuned coordination, from the programming language to the hardware, through the compiler and multiple layers of the run-time system. This document presents our work in *split* compilation and parallelization. Split compilation relies on automatically generated semantical annotations to enrich the intermediate format, decoupling costly offline analyses from lighter, online or just-in-time program transformations.

Our work focuses on automatic vectorization, a key optimization playing an increasing role in modern, power-efficient architectures. Our research platform uses GCC's support for the Common Language Infrastructure (CLI ECMA-335); this choice is motivated by the unique combination of optimizations and portability of GCC, through a semantically rich and performance-friendly intermediate format. Implementation is still in progress.

Keywords Performance Portability, Vectorization, Bytecode, CLI, CIL, Annotations

1. Introduction

The design of high-performance architectures is dominated by performance-per-Watt considerations, in particular in the embedded domain. This trend lends to massively parallel, heterogeneous processors, where specialized computing resources can be switched on or off depending on the application. This also lends to wide platform variability, from the higher to lower end of the market, and across different application domains (numerical and signaprocessing algorithms). This combined variability and heterogeneity challenges the adaptation of increasingly complex software. Meanwhile, mainstream languages are getting higher level, more abstract from the hardware and traverse several compilation steps.

In this context, performance portability is dramatically endangered. Just-in-time compilation, bytecode intermediate languages, and virtual machines have been proposed to tackle with this problem. The goal is to make the right optimization choice at the proper compilation/execution stage, bringing together the accuracy of dynamic analysis with the performance impact of static code generation. Preserving semantical information along the multiple compilation and run-time stages is necessary to get the best of this performance deal.

Our goal is to let the intermediate language carry relevant and generic performance information about parallelism. This allows to delay the key optimizations until future compilation stages. It unfortunately emphasizes the lack of generic way to split those optimizations along compilation stages, preserving semantical information across intermediate languages. Addressing this specific challenge led us to the concept of *split compilation*, where optimization passes are virtually extended across multiple program representations and compilation stages, structuring the flow of information while preserving (performance) portability.

Our work leverages on the recent extension of GCC to another virtual execution environment, the Common Language Infrastructure (CLI ECMA-335), the original execution environment of Microsoft .NET, also supporting independent and free software developments.

This extended abstract presents the design of bytecode annotations to pass semantical information from the offline to the just-in-time (JIT) stages of compilation. These annotations support automatic vectorization, starting from a fully portable bytecode. They carry information that is either

1. hard to retrieve from the intermediate language, like pointer aliasing;
2. too costly for the limited time budget of a JIT compiler, like array dependence analysis;
3. not usable in a target-independent optimization, for plain portability or performance portability issues, like the selection of target-specific vector instructions.

The paper is structured as follows. Section 2 presents the motivations for split compilation, in the context of vector parallelism. Next we present the baseline GCC infrastructure supporting our experiments, starting with the GCC vectorizer in Section 3 and the GCC CLI back-end and front-end in Section 4. Section 5 describes our approach, followed by some implementation status in Section 6.

2. Motivation

Portability has driven the definition of compact intermediate representations. This, in turn, has triggered advances in Link-Time and Just-in-Time compilation.

2.1 Why split compilation?

Program optimization combines static and/or dynamic analysis, pattern matching, code transformation and generation, performance modeling, and operation research algorithms. Typical optimization passes entangle these aspects very tightly, sometimes blurring the difference between them (e.g., analyses for legality or profitability). This entanglement is known to bring diminishing returns when composing compilation passes. It is also a serious source of information loss, leading to missed optimization opportunities when running multiple compilation stages.

Split compilation specifically addresses the design of optimization algorithms to break this entanglement. It relies on annotations of the intermediate languages, to virtually extend an optimization across multiple optimization stages. The big picture is summarized in Figure 1.

Annotations are a common way to pass additional information to the compiler. They are usually expressed in a declarative

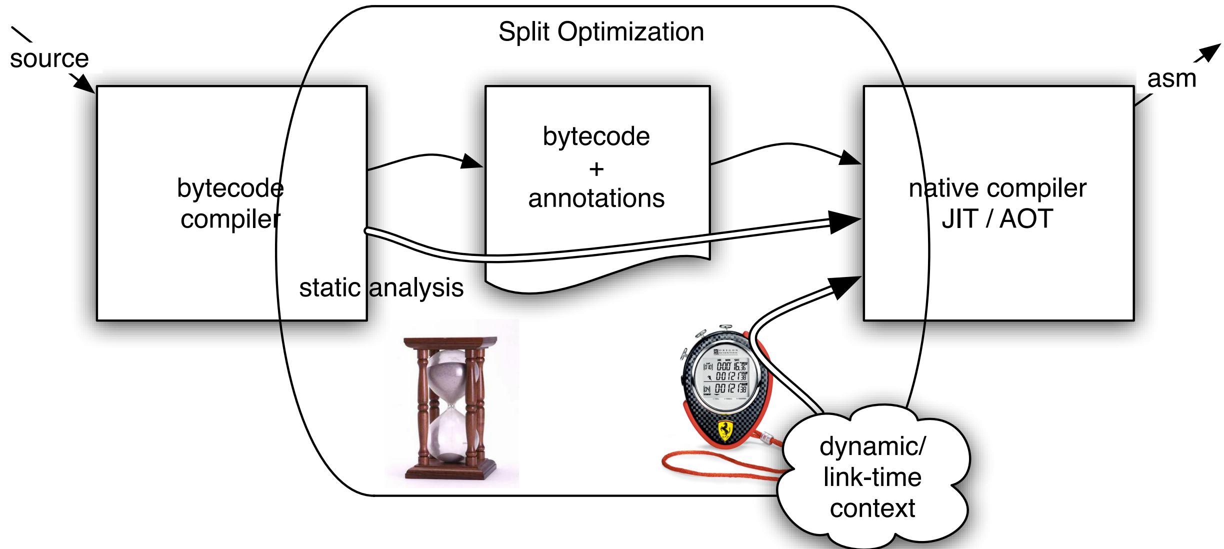

Figure 1. Split compilation framework

way (from correctness assertions to compilation or run-time hints), while preserving the genericity of the language/algorithm.

Our goals by using those language metadata are threefold:

- extended expressiveness of intermediate representation (parallelism);
- facilitate the mapping to specific hardware accelerators;
- abstract the result of an expensive off-line analysis.

In a JIT context, the size and portability of these annotations are the largest constraints. We are looking for split optimization designs satisfying two conflicting goals:

- minimize the amount of semantical information flowing across compilation stages; or minimize the size of this information, maximizing its density;
- minimize the performance hit due to the delayed analyses and transformations taking place at run-time;
- maximize the performance benefits to the generated code.

2.2 Related work

In general, annotation-driven compilation uses the extra information to hide overheads associated with managed language semantics, to refine the legality constraints for program transformation, as compilation hints to drive profitability heuristics, or simply to speedup online passes; see [8, 14, 9] for state-of-the-art Java techniques. Alternatively, Dittamo et al. [6] use manual annotations of high level programs to enable parallelization of annotated bytecode.

Link-time optimization [16, 10] and assembly-level optimization [1] are common applications of annotation-driven, multi-stage compilation; in this case, annotations are mostly used to carry missing static information.

Split compilation goes one step further, virtually extending program optimizations across multiple passes, compilation stages and intermediate languages. This implies revisiting optimization algorithms, to bring the most aggressive optimizations to dynamic code generators. Unlike compilation-time amortization schemes in classical JIT optimizers (profile- and profitability-driven), split compilation does *not* trade compilation time for generated code quality.

In some situations, annotations may appear in the form of scalar data (legality or profitability hints). Yet, for improved code compaction and transformation effectiveness, we also consider more general forms of metaprogramming. This would be necessary to support specialization (partial evaluation) and application-specific code generation; early experiments where conducted with MetaOcaml [2, 3, 7].

2.3 Two-stage vectorization

Intra-word vector instructions — also called SIMD instructions or extensions — have become pervasive in modern hardware. They are widely used in computation-intensive routines relying on hand-written assembly code or using compiler built-ins.

Auto-vectorization is a technique to select SIMD instructions through sophisticated static analyses, loop transformations and pattern-matching. It has made its way to production compilers [12, 17], although it is sensitive to fragile pattern matching rules at the moment. It is yet restricted to classical, static compilers, largely due to the compilation time and infrastructure it requires, and also due to portability constraints. For these reasons, it appears as an ideal target for split compilation.

Whereas LLVA proposes to extend bytecode representations to support intra-word vectors in JIT compilers [15], we do not accept to lose portability and prefer “how to vectorize” hints. We will drive vectorizing tranformations in an application- and processor-independent way, taking advantage of the extensibility of current intermediate languages through *bytecode metadata*.

To clearly identify the potential impact of our approach, we prefer to eliminate all overheads associated with managed languages like Java and C# (dynamic checks and typing, exception-handling, garbage collection): thanks to the CLI design and the GCC CLI implementation, we may start from unmanaged C sources. This choice also helps differentiating split compilation for annotation-driven overhead removal [14].

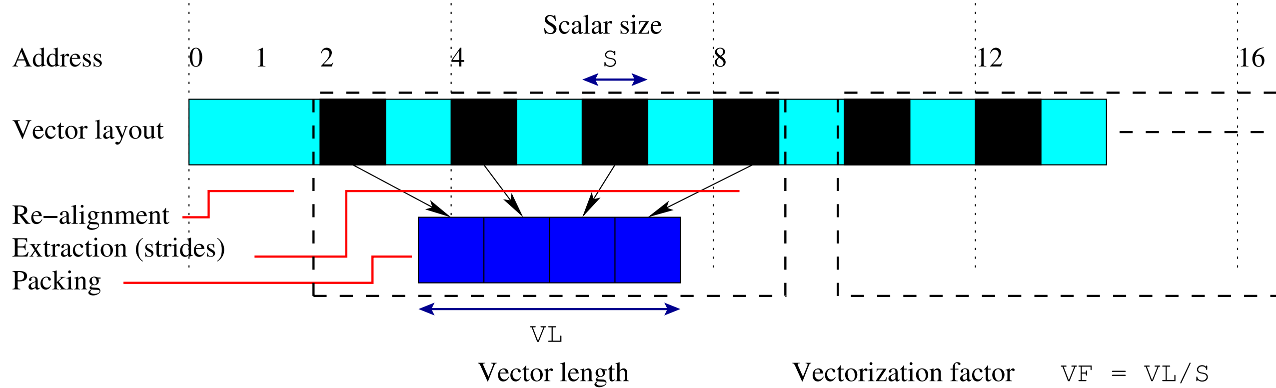

Figure 2. Vectorization principles

Beyond methodological motivations, this choice is also driven by STMicroelectronics's evaluation of CLI to run computation-intensive tasks on embedded platforms [5].

Of course, split compilation would also apply to code generation from higher level languages.

3. The GCC Vectorizer

Vectorization is a loop transformation performed in the GIMPLE-SSA (Tree-SSA) middle-end of the compiler [11, 12]. It consits of a static analysis phase, followed (if successful) by the actual code transformation. It is (currently) applied to innermost loops with a single basic block, and (currently) focuses on loop-induced vectorization (rather than block-induced).

Figure 2 summarizes the most important properties associated with intraword vectorization.

GCC performs several analysis steps to recognize vectorizable loops:

- it determines the data types to decuce the appropriate vector types, leading to the inference of the *vectorization factor* `VF` (number of loop iterations used in a vector operation);
- it characterizes access patterns for each memory reference across loop iterations; this analysis is needed for memory dependences, stride and alignment computation;
- pattern recognition discovers reduction, dot product, and other specific idioms;
- from array dependence analysis, it determines if some loop-carried dependences exist, and if so, if the associated dependence distance is lower than `VF` (pointer aliasing and pointer arithmetic are treated transparently through prior passes, and inductive scalar dependences are ignored);
- it builds interleaving groups of instructions (in case of strided access);
- it builds memory access patterns (simple increments or part of interleaving group);
- it checks if operations are supported by the target platform.

Then the vectorizer transforms the loop into a vectorized one:

- versioning and peeling the loop for alignment;
- peeling trailing loop iterations for divisibility with the `VF`;
- top-down scan (definitions before uses), inserting vector statements;
- generate loads/stores for interleaving groups, plus the relevant data reordering through even/odd extractions and high/low interleaving;
- transform loop bounds, and generate epilogue if the loop was peeled.

Eventually, GIMPLE vector tree nodes are transformed into native instructions of the target processor at the RTL level.

4. GCC for the CLI

In June 2006, a project aimed at the development of a back-end producing CLI-compliant binaries was born in the GCC community. One year later, the back-end supports C99 (with a few exceptions); it already delivers excellent results, both in terms of performance and code size [4].

4.1 About the Common Language Infrastructure

CLI (*Common Language Infrastructure*) is a framework that defines a platform independent format for executables and a run-time environment for the execution of applications. CLI was invented and it is still best known to be the foundation of Microsoft .NET framework.

CLI executables are encoded in a *Common Intermediate Language* (*CIL*), a machine-independent instruction set. This is possible since CIL is not bound to the instruction set of the machine on which applications are executed. Since a CLI application does not contain native code, it is not directly executable. A CLI virtual machine is required in order to run a CLI binary; a wide range of execution techniques are possible, which include interpretation, ahead-of-time and just-in-time compilation.

4.2 GCC CLI back-end

Unlike a typical GCC back-end, CLI back-end [5] stops the compilation flow at the end of the middle-end passes and, without going through any RTL pass, it emits CIL bytecode from GIMPLE representation. As a matter of fact, RTL is not a convenient representation to emit CLI code, while GIMPLE is much more suited for this purpose.

CIL bytecode is much more high-level than a processor machine code. For instance, there is no such a concept of registers or of frame stack; instructions operate on an unbound set of locals (which closely match the concept of local variables) and on elements on top of an evaluation stack. In addition, CIL bytecode is strongly typed and it requires high-level data type information that is not preserved across RTL.

4.2.1 Target machine model

CLI is seen as a target machine by GCC, its machine description presents the following properties.

- Languages as C and C++ need to know the size of pointers at compile time, and cannot defer this choice to CLI initialization time, so two separate 32 bit and 64 bit targets are defined.
- The CLI back-end deals with unmanaged data like C and C++, therefore two separate targets are defined where the size of the pointer is set to 32 (this is `cil32` target) or 64 (for `cil64`). Natural modes for computations go up to 64 bits.
- In the absence of a `packed` attribute, alignment rules specify that natural alignment is always followed. This avoids annotation bloat.
- Properties exclusively needed by RTL passes are skipped. This is a mere consequence of the CLI back-end operating on GIMPLE.
- Though the CLI back-end does not reach RTL passes, there is a minimum set of RTL-related description that must be present anyway. For instance, a few instruction selection patterns are mandatory, while others are used by some heuristics for cost estimation; there must be a definition of the register sets and a few peculiar registers have to be defined. As a rule of thumb, the machine model contains the simplest description for these properties, even if this makes little sense for CLI target.

Though most GIMPLE tree codes closely match what is representable in CIL, some simply do not. Those tree codes could still be expressed in CIL bytecode by a CIL-emission pass; however, it would be much more difficult and complicated to perform the required transformations at CIL emission time (i.e.: those that involve generating new local temporary variables, modifications in the control-flow graph or in types...), than directly on GIMPLE expressions.

Pass `simpcil` is in charge of performing such transformations. The input is any code in GIMPLE form; the outcome is still valid GIMPLE, it just contains only constructs for which the CIL emission is straightforward. Such a constrained GIMPLE format is referred as "CIL simplified" GIMPLE throughout this documentation.

Pass `simpcil` is performed just once, after leaving SSA form and immediately before CIL emission.

Transformations performed include expansion of GIMPLE instructions (comparisons, bit instructions, conditionals...) and data (arrays, bit fields, variable initialization, rename for global variables...) that are not well mapped to CIL.

4.2.2 CIL emission pass

Pass `cil` receives a CIL-simplified GIMPLE form as input and produces a CLI assembly file as output. It is the final pass of the compilation flow.

Before the proper emission, `cil` currently merges GIMPLE expressions in the attempt to eliminate local variables.

Aside from obvious translations, GIMPLE pointer nodes are translated to native `int`s, and all directly addressed (unmanaged) data — `structure` records, `unions`, `enumerates`, and especially arrays — are emitted as `valuetypes`, i.e., data types with explicit layout.[1]

To preserve C semantics which allow pointer arithmetics, arrays are represented by explicitely addressed `valuetypes`. Performance wise it is also important that that we do not use CIL managed arrays (garbage collected, with run-time checks) and this was a key element in the acceptance of CIL bytecode for computation-intensive embedded applications [5].

This feature also impacts our split compilation framework, as it clearly separates performance gains associated with the reduction of array access overhead from the expected vectorization gains. Of course, split vectorization does not need such unmanaged arrays to be applicable in general.

4.3 GCC CIL front-end

A CIL front-end, `Gcccil` [13] has also been implemented. Its first goal is to accept code generated by the CLI back-end, i.e. coming from a C source. To support more advanced CLI features such as reflection or garbage collection it will require a runtime library (like `libgcj`).

The GCC front-end for CIL does not implement its own CLR parser. Instead, it uses Mono to "parse" the input assembly. That is, Mono is used to load the assembly and parse the metadata and types. The front-end only has to parse the actual CIL code of each method.

CIL basic types (`int32`, `intptr`, `float` ...) are translated to their obvious GCC equivalent. To translate classes and value types, the `GCC_RECORD_TYPE` tree node is used.

`Gcccil` has also to deal with types with explicit layout and size as `valuetype`. Those are particularly important because unmanaged arrays generated by the CIL back-end are part of them. `Gcccil` took advantage of the existing support for explicit layout in ADA, even though it dos not cover all possible definitions. This is of particular importance for vectorization as the CLI back-end emits `valuetypes` for arrays.

Once the types have been parsed, *Gcccil* parses the CIL code stream for the methods defined in the assembly in order to build GCC GENERIC trees for them.

`Gcccil` cannot compile some methods if they use some unsupported features. In those cases, those methods can be skipped, allowing the user to provide a native implementation if necessary.

5. Design of Split Vectorization

We present here our work in progress in the split vectorizer which is at a prototype stage. It handles only a few constructs and is still closely tied to the GCC vectorizer implementation. We discuss afterwards our ideas to generalize the framework for more target independance.

5.1 Annotating of CIL intermediate language

CIL bytecode is our intermediate representation and we are using its metadata facilities to pass partial knowledge about vectorization.

In CIL, metadata are expressed as *custom attributes*. CIL custom attributes are structured, meaning they are typed object inheriting from the System.Attribute class. Nevertheless, attributes can be attached to fieds, types and methods, but *not* directly to code, meaning CIL instructions themselves or blocks of instructions. Annotations called *modopts* can also be attached to method arguments then modifying the method signature (e.g. with a restrict attribute).

Thus, so far annotations are straightforward structures generated by the vectorizer analysis and attached to methods. Loops are identified by the loop number, as identified by GCC's analysis of natural loops. Note that in a JIT it requires the analysis of natural loops. Figure 3 shows a simple exemple of an annotation of a loop which doesn't modify the original loop. This loop only requires a simple strip-mining scheme.

This simple scheme presents several issues :

- low level information on statements of the loop is lost (induction variables: loop stride to modify)

[1] Notice such a feature does not exist in Java. It is clearly a disaster in terms of portability, but was necessary to support C and C++.

```
[vectorizable(loop=1,vector_size=16)]
foo() {
  int a;
  int x[N],y[N],z[N];
  for(i=0; i<N; i++){
    z[i] = a*x[i] + y[i]
  }
}
```

Figure 3. simple annotation of a loop

- difficult to cope with more complicated code blocks
- information from the vectorizer is tied to the hardware

The first comes from the grammar of the intermediate representation itself: custom attributes cannot be attached to instructions themselves. A serious problem comes from the vectorization information for instructions inside the loops which could possibly have a different layout after their round trip outside of GCC.

The last issue though is closely linked with the vectorizer infrastructure in GCC. In particular, some analyses are tied to the hardware and have to be performed on the target machine:

- the data types supported on the target, i.e. sizes of the vectors;
- the vectorization factor depends on those types, and the packing/unpacking scheme;
- the data layout may be unknown, hence the alignment, but alignment constraints are target-dependent.

Both offline-analyzable and online-analyzable properties have been listed on Figure 2, except dependence distance information which is tied to the control flow.

In the case of a JIT compiler, delaying some analyses may have some advantages: additional dynamic information may help generate better code, e.g., with respect to data alignment.

Eventually, the difficulty is to reassign information from annotations to the proper data or control structure.

5.2 GCC as a JIT-compiler

For the sake of simplicity and prototyping, we are not using a JIT compiler but `Gcccil` as a special case of "ahead of time" compilation. In fact it is not totally irrealistic as some applications could be compiled at install-time using a dedicated compiler (e.g., media codecs on embedded hardware). CIL is really used as a portability layer.

In a real-world JIT compiler there would be several infrastructure differences:

- the need to implement some loop transformations (strip-mining, peeling) not found in JIT compilers;
- baseline code generation should be simpler and faster.

5.3 Compilation flow

Thanks to GCC's flexibility, our prototype split compilation framework consists of 2 versions of GCC, one offline to generate bytecode and one ahead-of-time to generate native code. Both use vectorization but annotation generation replaces the actual transformation in the bytecode compiler, whereas all expensive analyses are disabled in the native compiler.

Annotations are passed as custom attributes in CIL (outside GCC).

Inside GCC they are mirrored to GCC attributes, which are of the form `TREE_LIST` of tree nodes. They have less flexibility though, as they do not have a type system and follow a looser structure.

5.3.1 Compilation to CIL: offline stage

- Modified autovect pass (offline version).
  - The loop analysis relies on GCC's infrastructure for expensive operations like scalar evolution (induction variable recognition and substitution) and dependence analysis.
  - If vectorization is successful, it has to produce an attribute node in GCC with the annotation attached to the enclosing function and identified by with its loop number. We suppose there are neither *downstream* passes impacting the ordering and count of loops, nor *upstream* passes in the second stage native compiler resulting in skewed loop numbers.
  - Do not perform the actual loop transformation and code generation.
- The CIL code generation pass generates custom attributes from the attributes and additionnal data as classes describing the attributes.

5.3.2 Compilation from CIL: online stage

- Read CIL custom attributes and parse them into GCC attributes
- Modified autovect pass (online version).
  - Check if there exists an annotation corresponding to the current loop number.
  - Build vectorization structures from the annotation.
  - Complete the structures with missing target-specific information (very fast analyses).
  - Perform the actual loop transformation and generate the code.

In addition, the CLI front-end had to be slightly modified to comply with the single basic block and latch per loop constraint. A temporary hack permits to recognize arrays, using the mangled name produced by the back-end.

5.4 Describing fine grain parallelism

One of the main goals of our vector-aware IL is to address different targets. Rather than a particular vector size, we are interested to know the maximal vector length for the code. Thus we need our annotations to use a higher level representation, i.e. a virtual vector notation á la Fortran90 for array statements.

Unfortunatly annotations can only be attached to metadata, not blocks of code. This is a challenge for the expression of fine grain parallelism. We considered the following solutions:

- Having a library with special (virtual) vector types and intrinsics and emitting a vectorized version.
- Using method level annotations. This results in some code bloat though: making the link between annotation and code is not trivial in this scheme and requires hand made mapping using offsets, special numbering.
- In the previous scheme, loops bodies can be outlined which gives more flexibility to annotate them. It relies heavily on the inliner and results in code growth as it requires to pass all local variables as arguments. Arguments can be annotated thanks to standard attributes called `modopts`.
- last solution is to annotate data uses. Locals can not be annotated, and making everything global kills optimization opportunities in the JIT so the best way is to annotate types: we flag when a particular type access has a "special" meaning, and annotate the semantics of those accesses in types as seen in Figure 4.

```
foo() {
  [vector ->] int x[N]; /* vector access */
  [vector ->] int y[N];
  [vector ->] int a[1]; /* promotoed to vector type */
 ...
  for(i=0; i<N; i++){
      z.vec[i] = a.vec * x.vec[i] + y.vec[i];
   }
}
```

Figure 4. using 'vector' array types

In practice, as shown in Figure 5, we combine custom attributes to annotate type and access semantics with properties of value types, namely explicit layout, giving a way to access to the same data with a different name (similar to C unions, as there is no type aliasing in CIL). The data access is still correct, with a different type of indirection to access the variable (loading the array address with `ldsflda` and then loading the field with `ldfld` instead of `ldind`, which the JIT should turn into a constant `0` plus the address).This scheme presents an overhead for true scalar variables as they are turned into value types.

```
.class public explicit sealed serializable
    ansi 'array' extends
    ['mscorlib']System.ValueType
{
 .size 64
 .field [0] public specialname
    int32 'elem__'

 // alternate vector access via union-like fields
 .field [0] public specialname
    int32 'vec_elem__'
 .custom instance void
    class 'VirtualVector'::ctor() = /* ... */
}

...
 // load the array adress
 ldloca.s 0
// load the field :
 ldfld int32 'array'::vec_elem__
...
```

Figure 5. bytecode for 'vector' arrays

This kind of annotation would also enable to encode more complicated patterns as shown in Figure 6 and add a special semantics to those new data accesses which are statically encoded in the type. Data shuffling can introduce intermediate copies in this representation though which can lead to bytecode growth.

It can be noted that this scheme also allows the use of some methods added to the value type 'array' class annotated with special 'intrinsic' semantics for the compiler, similarly to a library scheme.

6. Implementation Status

Work is still in progress. Only simple cases could be vectorized so far, and we did not collect significant performance mesurements with pre-vectorizer passes disabled.

We expect to show significant performance enhancements in terms of compilation time in the online pass. We also expect to demonstrate performance levels on par with static (one-stage) compilation of the source benchmarks. Our optimism is driven by the good scalar performance results by Costa et al. with `Gcccli`. [5].

```
foo() {
 [vector ->] int c[1]; /* promoted to vector type */
 [vector ->] int a[N];
 for(i=0; i<N; i++){
    c.vect_reduction += a.vect[i];
 }

 for(i=0; i<len; i++){
    ... =  a.vec_even[2*i]
 }
}
```

Figure 6. more complex patterns

7. Perspectives

Beyond automatic intra-word vectorization, the next step will be to map general purpose intermediate languages such as CIL to parallel heterogeneous targets, e.g, graphical processing engines (GPU) or numerical computation accelerators.

Our ongoing work in split compilation also extends to register allocation. The fast linear-scan algorithm of most JIT compilers can be enhanced thanks to annotations, and driven to perform as well as any static allocation and spilling heuristics. The idea here is not only to feed an existing algorithm with annotations, but to revisit the whole process, and to propose a generic allocation and spilling framework where annotations allow to emulate costly variants in linear time. The same principle could also be applied to loop transformations or to coarser grain function level parallelism.

In the long term, we also plan to integrate split compilation in a link-time optimization framework. In synchrony with GCC's LTO effort, we will suggest intermediate language improvements in favor of future split compilation uses.

Acknowledgments

We would like to thank Sebastian Pop for his help and advice. We are also indebted to the autovect group at IBM Haifa and to Roberto Costa. Eventually, split compilation and GCC CLI were born from the original vision and dedicated efforts of Marco Cornero from STMicroelectronics.

References

[1] F. Bodin, E. Rohou, and A. Seznec. Salto: System for assembly-language transformation and optimization. In *Workshop on Compilers for Parallel Computers (CPC'96)*, Dec. 1996.

[2] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In *GPCE'03*, volume 2830 of *Lecture Notes in Computer Science*, pages 57–76. Springer-Verlag, 2003.

[3] A. Cohen, S. Donadio, M.-J. Garzaran, C. Herrmann, O. Kiselyov, and D. Padua. In search of a program generator to implement generic transformations for high-performance computing. *Science of Computer Programming*, 62(1):25–46, Sept. 2006. Special issue on the First MetaOCaml Workshop 2004.

[4] M. Cornero, R. Costa, R. F. Pascual, A. C. Ornstein, and E. Rohou. An Experimental Environment Validating the Suitability of CLI as an Effective Deployment Format for Embedded Systems. In *International Conference on High Performance Embedded Architectures & Compilers (HiPEAC)*, Göteborg, Sweden, Jan. 2008.

[5] R. Costa, A. C. Ornstein, and E. Rohou. CLI back-end for GCC. In *GCC Summit*, Ottawa, Canada, June 2007.

[6] C. Dittamo, A. Cisternino, and M. Danelutto. Parallelization of C# programs through annotations, 2007.

[7] S. Donadio, J. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. Garzaran, D. Padua, and K. Pingali. A language for the compact representation of multiple program versions. In *Languages and Compilers for Parallel Computers (LCPC'05)*, Lecture Notes in Computer Science, Hawthorne, New York, Oct. 2005. Springer Verlag. 15 pages.

[8] J. Jones and S. Kamin. Annotating java bytecodes in support of optimization. *J. of Concurrency: Practice and Experience*, 2000.

[9] C. Krintz and B. Calder. Using annotations to reduce dynamic optimization time. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–167, 2001.

[10] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *ACM Conf. on Code Generation and Optimization (CGO'04)*, San Jose, CA, Mar. 2004.

[11] D. Naishlos. Autovectorization in GCC. *GCC summit*, June 2004.

[12] D. Nuzman and A. Zaks. Autovectorization in GCC - two years later. *GCC summit*, June 2006.

[13] R. F. Pascual. GCC CIL frontend. http://www.hipeac.net/node/823, 2006.

[14] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing java using attributes. In *Compiler Construction, 10th International Conference (ETAPS/CC'01)*, pages 334–554, 2001.

[15] J. Robert L. Bocchino and V. S. Adve. Vector LLVA: a virtual vector instruction set for media processing. In *VEE '06: Proceedings of the second international conference on Virtual execution environments*, pages 46–56, New York, NY, USA, 2006. ACM Press.

[16] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the 2005 IEEE International Symposium On Signal Processing And Information Technology*, pages 7–12, Athens, 12 2005. IEEE.

[17] P. Wu, A. E. Eichenberger, A. Wang, and P. Zhao. An integrated simdization framework using virtual vectors. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 169–178, New York, NY, USA, 2005. ACM Press.