Proceedings of the 2nd International Workshop on GCC Research Opportunities (GROW'10)

co-located with HiPEAC'10

Pisa, Italy
January 23rd, 2010

http://cTuning.org/workshop-grow10

Table of contents:

- **Workshop foreword** **3**
  Dorit Nuzman[1] and Grigori Fursin[2]
  [1] IBM Haifa, Israel
  [2] INRIA, France

- **GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation** **4**
  Konrad Trifunovic [2], Albert Cohen [2], David Edelsohn [3], Li Feng [6], Tobias Grosser [5], Harsha Jagasia [1], Razya Ladelsky [4], Sebastian Pop [1], Jan Sjödin [1], and Ramakrishna Upadrasta [2]
  [1] AMD, USA
  [2] INRIA Saclay and Paris-Sud 11 University, France
  [3] IBM T. J. Watson Research, USA
  [4] IBM Haifa Research, Israel
  [5] University of Passau, Germany
  [6] Xi'an Jiaotong University, China

- **Extending GCC with a multi-grain parallelism adaptation framework for MPSoCs** **20**
  Nicolas BENOIT and Stephane LOUISE
  CEA LIST, France

- **A case study: optimizing GCC on ARM for performance of libevas rasterization library** **34**
  Dmitry Melnik [1], Andrey Belevantsev[1], Dmitry Plotnikov[1], and Semun Lee[2]
  [1] ISP RAS, Russia
  [2] Samsung, Korea

- **Portable and Efficient Auto-vectorized Bytecode: a Look at the Interaction between Static and JIT Compilers** **47**
  Erven Rohou
  INRIA, France

- **Compiler-controlled and Compiler-hinted Voltage Scaling Approaches** **61**
  Dmitry Zhurikhin [1], Andrey Belevantsev [1], Kirill Batuzov [1], Valery Ignatiev [1], Roman Zhuykov [1], and Semun Lee [2]
  [1] ISP RAS, Russia
  [2] Samsung, Korea

- **Using Software Metrics to Evaluate Static Single Assignment Form in GCC** **73**
  Jeremy Singer [1], Christos Tjortjis [2], and Martin Ward [3]
  [1] University of Manchester, UK
  [2] University of Ioannina, Greece
  [3] De Montfort University, UK

- **A New Intermediate Representation for GCC based on the XARK Compiler Framework** **89**
  Jose M. Andion, Manuel Arenaz, and Juan Tourino
  University of A Coruna, Spain

- **Transforming GCC into a research-friendly environment: plugins for optimization tuning and reordering, function cloning and program instrumentation** **101**
  Yuanjie Huang [1,2], Liang Peng [1,2], Chengyong Wu [1], Yuriy Kashnikov [4], Jörn Renneke [3], Grigori Fursin [3]
  [1] ICT, Chinese Academy of Sciences, China
  [2] Graduate School of the Chinese Academy of Sciences, China
  [3] INRIA Saclay, France
  [4] University of Versailles at Saint-Quentin-en-Yvelines, France

Workshop foreword:

Welcome to the Second International Workshop on GCC Research opportunities (GROW'10). This workshop series is intended to bring together people from industry and academia that are interested in conducting research based on GCC and enhancing this compiler suite for research needs. The workshop will promote and disseminate compiler research with GCC, as a robust industrial-strength vehicle that supports free and collaborative research.

This year, we received 15 submissions; each was evaluated by four members of the program committee. Eight of the submitted papers were accepted, based on their quality and focus, for presentation at the workshop. Besides the papers, the program this year also includes an invited talk and panel, on future research and development directions of GCC.

We hope that this year's attendees will find the ideas presented in the papers and the panel discussions interesting and useful. Best wishes for a productive meeting!

Grigori Fursin, Dorit Nuzman
GROW'10 organizers

GRAPHITE Two Years After
First Lessons Learned From Real-World Polyhedral Compilation

Konrad Trifunovic[2], Albert Cohen[2], David Edelsohn[3], Li Feng[6], Tobias Grosser[5], Harsha Jagasia[1], Razya Ladelsky[4], Sebastian Pop[1], Jan Sjödin[1], and Ramakrishna Upadrasta[2]

[1] Open Source Compiler Engineering, AMD, Austin, Texas, USA
{harsha.jagasia, sebastian.pop, jan.sjodin}@amd.com
[2] INRIA Saclay – Île-de-France and LRI, Paris-Sud 11 University, Orsay, France
{albert.cohen, konrad.trifunovic, ramakrishna.upadrasta}@inria.fr
[3] IBM T. J. Watson Research, Yorktown Heights, USA
dje@watson.ibm.com
[4] IBM Haifa Research, Haifa, Israel
razya@il.ibm.com
[5] University of Passau, Passau, Germany
grosser@fim.uni-passau.de
[6] Xi'an Jiaotong University, Xi'an, China
nemokingdom@gmail.com

Abstract. Modern compilers are responsible for adapting the semantics of source programs into a form that makes efficient use of a highly complex, heterogeneous machine. This adaptation amounts to solve an optimization problem in a huge and unstructured search space, while predicting the performance outcome of complex sequences of program transformations. The polyhedral model of compilation is aimed at these challenges. Its geometrical, non-inductive semantics enables the construction of better-structured optimization problems and precise analytical models. Recent work demonstrated the scalability of the main polyhedral algorithms to real-world programs. Its integration into production compilers is under way, pioneered by the GRAPHITE branch of the GNU Compiler Collection (GCC). Two years after the effective beginning of the project, this paper reports on original questions and innovative solutions that arose during the design and implementation of GRAPHITE.

1 Introduction

Despite several decades of research into the Polyhedral model, there is still no general-purpose production compiler using the Polyhedral model internally. The situation is changing with the demonstration of the scalability of polyhedral algorithms and with the widespread dissemination of multicore processors and hardware accelerators. Two proprietary polyhedral compilers are in development: the R-Stream compiler from Reservoir Labs [26], and IBM's polyhedral extension of its XL compiler suite [35].

This paper describes the GRAPHITE compilation pass of GCC, embedding polyhedral analyses and transformations into the GNU Compiler Collection (GCC) [32]. Polyhedral information is extracted directly from the GIMPLE intermediate representation, in three-address, *Static Single Assignment* (SSA) form. This is a major difference with traditional source-to-source polyhedral compilers which operate on high-level abstract syntax. Operating directly on the three-address code brings in new challenges but also new opportunities: we can leverage existing analyses in the compiler and interact with a wealth of optimizations.

GRAPHITE is based on the polyhedral representation designed by Girbal et al. [13]. This rich algebraic representation enables the composition of polyhedral generalizations of classical loop transformations, decoupling them from the syntactic form of the program. Classical transformations like loop fusion or tiling can be composed in any order and generalized to imperfectly-nested loops with complex domains, without intermediate translation to a syntactic form (avoiding code size explosion). GRAPHITE also aims at providing precise performance models and profitability prediction heuristics. Its applications include automatic parallelization and vectorization, offloading of computational kernels onto hardware accelerators, memory hierarchy usage optimizations, cost modelling and static code analysis (e.g., static debugging of parallel programs).

The paper is structured as follows. Section 2 discusses related work. Section 3 describes the design of GRAPHITE. Section 4 explores the current optimizations implemented in GRAPHITE. Representation of pointer accesses is an original issue for polyhedral compilation and is presented in Section 5, before the conclusion in Section 6.

2 Related work

There have been many efforts in designing an advanced loop nest transformation infrastructure. Most loop restructuring compilers introduced syntax-based models and intermediate representations. ParaScope [10] and Polaris [6] are dependence based, source-to-source parallelizers for Fortran. KAP [19] is closely related to these academic tools.

SUIF [16] is a platform for implementing advanced compiler prototypes. PIPS [17] is one of the most complete loop restructuring compiler, implementing polyhedral analyses and transformations (including affine scheduling) and interprocedural analyses (array regions, alias). Both of them use a syntax tree extended with polyhedral annotations, but not a unified polyhedral representation.

The MARS compiler [28] unifies classical dependence-based loop transformations with data storage optimizations. However, the MARS intermediate representation only captures part of the loop information (domains and access functions): it lacks the characterization of iteration orderings through multidimensional affine schedules.

The first thorough application of the polyhedral representation was the Petit tool [20], based on the Omega library [23]. It provides space-time mappings for iteration reordering, and it shares our emphasis on per-statement transformations, but it is intended as a research tool for small kernels only. We also use a code generation technique that is again significantly more robust than the code generation in Omega [4].

Semi-automatic polyhedral frameworks have been designed as building blocks for compiler construction or (auto-tuned) library generation systems [21, 9, 39, 8, 36]. They do not define automatic methods or integrate a model-based heuristic to construct profitable optimization strategies.

The GRAPHITE project was first announced by Pop et al. in 2006 [32] but real development work started only one year later: the number of changes committed to the GRAPHITE branch are presented in Figure 2. The design of GRAPHITE is largely borrowed from the WRaP-IT polyhedral interface to Open64 and its URUK loop nest optimizer [13]. The CHiLL project from Chen et al. revisited the URUK approach focusing on source-to-source transformation scripting [8, 36]. Unlike URUK and CHiLL, GRAPHITE aims at complete automation, possibly resorting to iterative search or statistical modeling of the profitability of program transformations. Besides, unexpected

design and implementation issues have arisen, partly due to the design of GCC itself, but mostly due to the integration of the polyhedral representation in a general-purpose compilation flow, such as pointers, profile data, debugging information, resource usage (compilation time), pass ordering, interaction among passes, etc.

3 Design

The polyhedral analysis and transformation framework called GRAPHITE is implemented as a pass in the GNU Compiler Collection compiler. The main task of this pass is to: extract the *polyhedral model* representation out of the GCC three-address GIMPLE representation, perform the various optimizations and analyses on the polyhedral model representation and to regenerate the GIMPLE three-address code that corresponds to transformations on the polyhedral model. This three stage process is the classical flow in polyhedral compilation of source-to-source compilers [13, 7]. Because the starting point of the GRAPHITE pass is the low-level three-address GIMPLE code instead of the high-level syntactical source code, some information is lost: the loop structure, loop induction variables, loop bounds, conditionals, data accesses and reductions. All of this information has to be reconstructed in order to build the polyhedral model representation of the relevant code fragment.

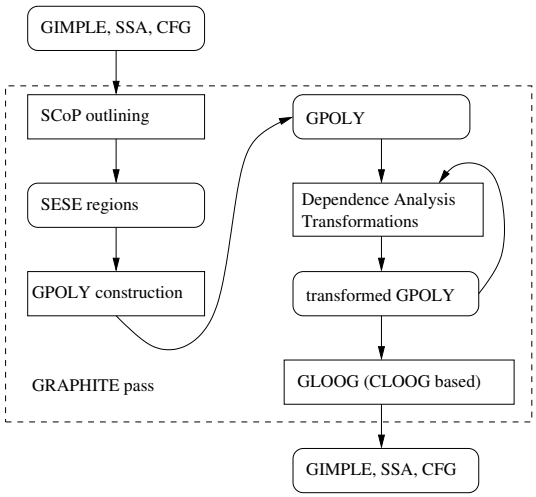


Fig. 1. Stages of the GRAPHITE pass

Year	Commits
2009	497
2008	216
2007	36
2006	10

Fig. 2. Changes to the GRAPHITE branch

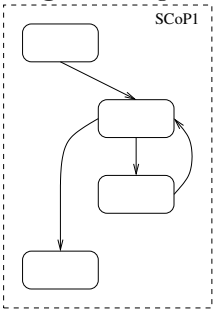


Fig. 3. Full SCoP

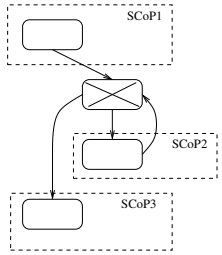


Fig. 4. Split SCoP

Figure 1 shows the stages inside the GRAPHITE pass: (1) the *Static Control Parts* (SCoPs) are outlined from the control flow graph, (2) polyhedral representation is constructed for each SCoP (GPOLY construction), (3) data dependence analysis and transformations are performed (possibly multiple times), and (4) GIMPLE code corresponding to transformed polyhedral model is regenerated (GLooG). The details of each stage are given in the following subsections.

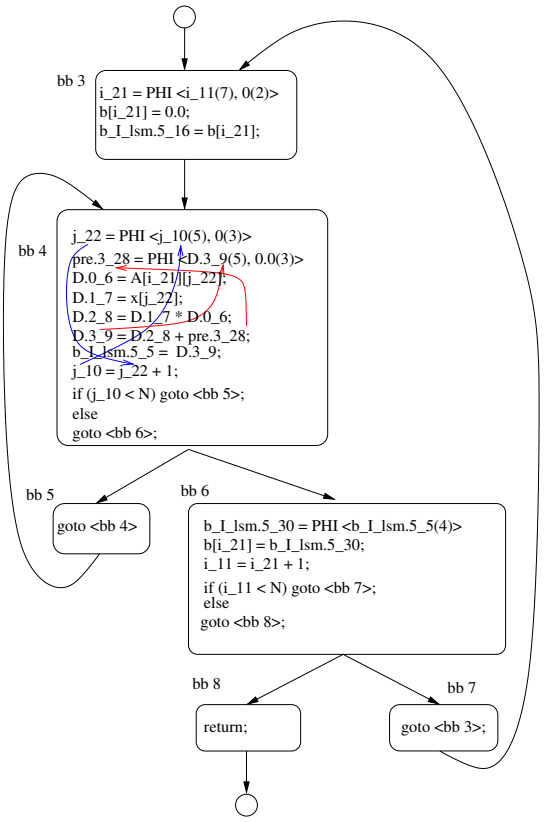


Fig. 5. GIMPLE code with CFG.

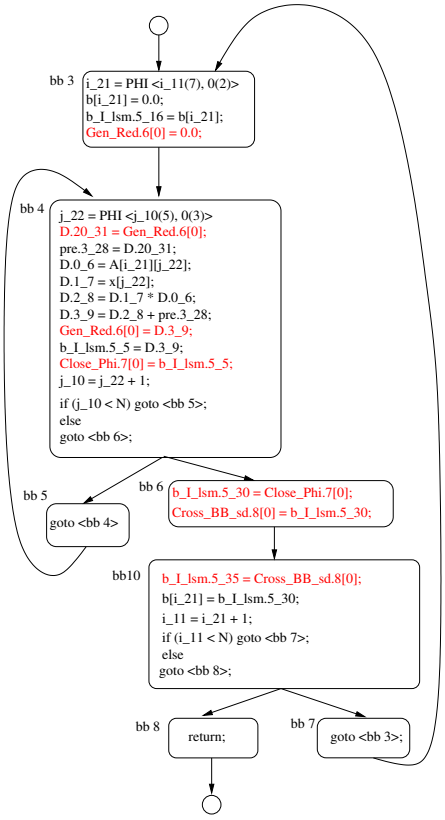


Fig. 6. Single-element arrays inserted to handle scalar dependences and reductions

Listing 1.1. Matvect

```
for (i = 0; i < N; i++) {
  b[i] = 0;
  for (j = 0; j < N; j++)
    b[i] += A[i][j] * x[j];
}
```

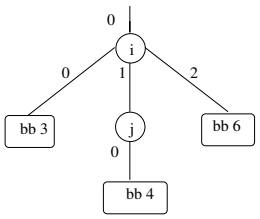


Fig. 7. LST tree

$$\mathcal{D}^S_{bb3} = \{(i) \mid 0 \le i \le N-1\}$$
$$\mathcal{D}^S_{bb4} = \{(i,j) \mid 0 \le i \le N-1 \wedge 0 \le j \le N-1\}$$
$$\mathcal{D}^S_{bb6} = \{(i) \mid 0 \le i \le N-1\}$$
$$\mathcal{F}_{dr1} = \{(i,a,s_1) \mid a = 0 \wedge s_1 = i \wedge 0 \le s_1 \le N-1\}$$
$$\mathcal{F}_{dr2} = \{(i,j,a,s_1) \mid a = 1 \wedge s_1 = j \wedge 0 \le s_1 \le N-1\}$$
$$\mathcal{F}_{dr3} = \{(i,j,a,s_1,s_2) \mid a = 2 \wedge s_1 = i \wedge s_2 = j \wedge 0 \le s_1, s_2 \le N-1\}$$
$$\mathcal{F}_{dr4} = \{(i,a,s_1) \mid a = 0 \wedge s_1 = i \wedge 0 \le s_1 \le N-1\}$$
$$\theta_{bb3} = \{(i,t_1,t_2,t_3) \mid t_1 = 0 \wedge t_2 = i \wedge t_3 = 0\}$$
$$\theta_{bb4} = \{(i,j,t_1,t_2,t_3,t_4,t_5) \mid t_1 = 0 \wedge t_2 = i \wedge t_3 = 1 \wedge t_4 = j \wedge t_5 = 0\}$$
$$\theta_{bb6} = \{(i,t_1,t_2,t_3) \mid t_1 = 0 \wedge t_2 = i \wedge t_3 = 2\}$$

Fig. 8. Components of the polyhedral representation of the GIMPLE code

3.1 SSA based SCoP outlining

The scope of the polyhedral program analysis and manipulation is a sequence of loop nests with constant strides and affine bounds. It includes non-perfectly nested loops and conditionals with boolean expressions of affine inequalities.

The maximal *Single-Entry Single-Exit* (SESE) region of the *Control Flow Graph* (CFG) that satisfies those constraints is called a *Static Control Part* (SCoP) [13, 7]. GIMPLE statements belonging to the SCoP should not contain calls to functions with side effects (`pure` and `const` function calls are allowed) and the only memory references that are allowed are accesses through arrays with affine subscript functions.

Since the GRAPHITE pass is scheduled at the stage where three-address code is in Static Single-Assignment form, all the analyses based on the SSA are available for use in GRAPHITE. This is crucial: in order to perform SCoP outlining the *scalar evolution* analysis framework of GCC is used [33]. Scalar evolution relies on SSA form to compute closed form expressions for induction variables. These closed forms are represented by structures called *CHains of RECurrences* (CHREC).

Chains of recurrences might represent induction variables (loop induction variables, array access subscript functions) that are affine or non-affine. For example, the scalar evolution of the `j_22` variable in the basic block 4 (Figure 5) is expressed as follows: $\{0, +, 1\}_2$, meaning that the starting value of the induction variable is 0, and it is incremented by 1 in each iteration of the loop number 2 (loop corresponding to basic blocks 5 and 6).

SCoP outlining proceeds as follows: first, a new SCoP region is opened, and then the basic blocks of the CFG are scanned in the dominator order. If the basic block contains a statement that is not representable in the polyhedral model then the whole basic block is deemed *difficult* so the current SCoP region is closed and a new SCoP is opened at the basic block dominated by the *difficult* basic block. Figure 3 shows one SCoP containing all the basic blocks, whereas Figure 4 shows how the *difficult* statement causes multiple SCoPs to be formed.

There exist limitations to the SCoP detection algorithm currently implemented in GRAPHITE: for example, determining whether the scalar evolution of a variable could be handled in the polyhedral model, or whether the variable should be considered a parameter of the SCoP. We think that a detection of SCoPs based on the structured CFG with SESE regions [18] would be more appropriate. A structural SCoP detection traverses the regions tree starting from the outermost SESE region, and tries to prove that all the statements in that maximal region can be handled in the polyhedral representation. When a *difficult* statement is detected, the statement is analyzed in all the regions containing it, from the outermost region to the innermost one, until either the statement is simple enough in a smaller region, or the region is the statement itself, in which case the statement cannot be handled at all. We consider the integration of a structural SCoP detection algorithm in future versions of GCC.

3.2 Construction of the polyhedral representation

Once the SCoPs are outlined, the polyhedral information is built for each basic block contained in a SCoP. The polyhedral representation consists essentially of three components: iteration domains, schedules, and data accesses.

The polyhedral information attached to each basic block in a SCoP is internally called GPOLY. All the components of the polyhedral model are represented as a system of affine equalities or inequalities, and for that purpose a *polyhedral library* is used. Currently, the Parma Polyhedra Library (PPL) [3] is used, but the representation is designed to accommodate other similar libraries.

Once again, the *scalar evolution* analysis framework is used to deduce the affine form of the loop bounds and global parameters (to build the iteration domains), memory addressing expressions (for the data accesses). Initial scheduling functions for each basic

block are deduced from the *Loop Statement Tree* (LST) showing the relative ordering of the basic blocks, and initial nesting structure of the loops. An example of the LST is given in Figure 7. More details explaining all the components of the polyhedral model are given in the following subsection.

Contrary to source-to-source polyhedral compilers [17, 31, 25], we have chosen to represent the schedules and the domains on a per *Basic Block* (BB) granularity instead of on a per statement granularity. This choice is somewhat rigid, since it prevents the independent scheduling of GIMPLE statements belonging to the same basic block. On the other hand, constructing the polyhedral representation per basic block might provide greater scalability. SCoP control and data flow are represented with three components of the polyhedral model [13, 7, 34]:

Iteration domains capture the dynamic instances of all basic blocks — all possible values of surrounding loop induction variables — through a set of affine inequalities. Each dynamic instance of a basic block S is denoted by a pair $(S, \mathbf{i})$ where $\mathbf{i}$ is the *iteration vector* containing values for the loop induction variables of the surrounding loops, from outermost to innermost. The dimension of iteration vector $\mathbf{i}$ is d^S. If the basic block S belongs to a SCoP then the set of all iteration vectors $\mathbf{i}$ relevant for S can be represented by a polytope: $\mathcal{D}^S = \left\{\mathbf{i} \mid \mathrm{D}^S \times (\mathbf{i}, \mathbf{g}, 1)^T \geq \mathbf{0}\right\}$ which is called the *iteration domain* of S, where $\mathbf{g}$ is the vector of *global parameters* whose dimension is $d_{\mathbf{g}}$. Global parameters are invariants inside SCoP, but their values are not known at compile time (parameters representing loop bounds for example).

Data references capture the memory locations of array data elements on which GIMPLE statements operate. In each SCoP, by definition, the memory accesses are performed through array data references. A scalar variable can be seen as a zero-dimensional array (array with only one dimension and only one element `A[0]`). Each array data reference (`data_reference`) inside a basic block is wrapped inside a `poly_dr` structure which contains the *data reference polyhedron*. The data reference polyhedron $\mathcal{F}$ encodes the *access relation* mapping iteration vectors in $\mathcal{D}^S$ to the array subscripts represented by the vector $\mathbf{s}$: $\mathcal{F} = \left\{(\mathbf{i}, a, \mathbf{s}) \mid \mathrm{F} \times (\mathbf{i}, a, \mathbf{s}, \mathbf{g}, 1)^T \geq \mathbf{0}\right\}$. The alias set number a captures points-to information (pointer aliasing); it allows to represent accesses through arbitrary pointers and will be defined in Section 5.

In contrast to classical polyhedral model representation [11, 22] we have chosen to represent *data references as relations*. This means that there is no one-to-one correspondence between iteration vector and subscript. This enables us to represent *memory regions* – when the data reference information is not complete (coming from interprocedural analysis for example). Nevertheless, very often, the correspondence between iterator vectors and data reference subscripts is a functional affine mapping $\mathbf{s} = f(\mathbf{i}, \mathbf{g})$.

In previous literature, the problem of link between dependence analysis and the analyses preceeding it, like alias analysis, has not been explored. This leads to inefficiency and impreciseness in representation, which are further exacerbated by software engineering constraints like modularity and portability. In Section 5, we will see a discussion of the problem and its algorithmic characterization as a hard combinatorial problem.

Scheduling functions are also called scattering functions inside GRAPHITE following CLooG's terminology. While iteration domains define the set of all dynamic instances for each basic block, they do not describe the execution order of those instances. In order to define the execution order we need to give to each dynamic instance the

execution time (date) [11, 22]. This is done in GRAPHITE by constructing the *scattering polyhedron* representing the relation between iteration vectors and time stamp vector $\mathbf{t}$: $\theta = \left\{(\mathbf{t}, \mathbf{i}) \mid \Theta \times (\mathbf{t}, \mathbf{i}, \mathbf{g}, 1)^T \geq \mathbf{0}\right\}$.

Dynamic instances are executed according to the lexicographical ordering of time-stamp vectors. By changing the scattering function, we can reorder the execution order of dynamic iterations, thus performing powerful *loop transformations*. More details on the transformations are given in Subsection 3.5.

Given the example GIMPLE code in Figure 5, the components of the polyhedral model representation are given in Figure 8.

3.3 Dependence analysis

In order to represent the semantics of the original program in the polyhedral model the dependence between dynamic instances of statements needs to be represented. The dependences are necessary to guarantee the correctness of loop transformations.

We are considering *data dependences* coming from the reads and writes of array elements. By definition [38], there is a data dependence from the dynamic instance of a basic block $(S_i, \mathbf{i}_{S_i})$ to the dynamic instance of basic block $(S_j, \mathbf{i}_{S_j})$ if both iteration vectors belong to their respective iteration domains (the execution is feasible), both instances refer to the same memory location and at least one of the data references is write and the instance $(S_i, \mathbf{i}_{S_i})$ is executed before $(S_j, \mathbf{i}_{S_j})$.

The *Polyhedral Dependence Analysis* (PDA) implemented in GRAPHITE is an *instance-wise dependence analysis* – meaning that the dependences are represented as polyhedra encoding the dependence relations between basic block instances. If projected to the Cartesian product of two iteration domains [38, 7, 34], the polyhedron encodes the iteration of the source of the dependence and the iteration of the sink of the dependence:

3.4 Handling scalar dependences

While the classical dependence analysis in the source-to-source polyhedral compilers considers only the data dependences between arrays (treating scalars as zero-dimensional arrays), this approach is not the most appropriate in the context of three-address code in the SSA form. If we are not considering scalar dependences, we are not capturing all the semantical constraints of the program – the transformed code could be illegal. If we are to convert all scalars to zero-dimensional arrays we would greatly increase the compilation time (polyhedral dependence check is algorithmically costly) and produce inefficient code.

The approach taken in GRAPHITE framework is to classify the scalar dependences into the following categories:

Intra basic block dependences occur between scalars inside a basic block. Those dependences are not considered by PDA in GRAPHITE. Since the statements inside the basic block cannot be rescheduled, the scalar dependences between statements inside a same basic block are not affected by polyhedral transformations. Those dependences are captured by use-def chains of the SSA representation.

Cross basic block dependences occur between scalars belonging to two different basic blocks. Those scalars are rewritten into zero-dimensional (single-element) arrays, such that PDA considers them as the regular array accesses. An example is given in Figure 6, where new zero-dimensional arrays (called `Cross_BB_sd`) are introduced.

Reduction dependences occur in data flow cycles that contain associative and commutative operations, like an accumulator variable performing a summation over the values of an array. For the regular reductions the new zero-dimensional arrays are introduced (as seen in Figure 6, where `Gen_Red` and `Close_Phi` arrays are introduced). If the reduction operator can be proved to be commutative and associative, then the dependences are marked as belonging to such a reduction. The former enables the optimizations, since the reduction operations can be rescheduled, disregarding the data dependences, if proved to be associative and commutative.

3.5 Transformations

According to the compositional approach of polyhedral transformations [13], the composition of multiple loop transformations in the polyhedral model can be expressed as a single scheduling transformation. By modifying the scheduling relations θ for each basic block, and regenerating the GIMPLE code according to those new schedules, we are able to perform arbitrary rescheduling of the basic blocks inside a SCoP.

In order to preserve the legality of the transformations, the *legality check* is performed for each data dependence relation.

Given the original data dependence relation $\mathcal{P}_{(S_i,R_k) \rightarrow (S_j,R_l)}$ representing the pairs of iterations which need to be executed in the specific order, the other polyhedron $\mathcal{P}'_{(S_i,R_k) \rightarrow (S_j,R_l)}$ is computed, giving those pairs of iterations that are violating the original dependence (they are executed in reversed order according to the new schedule). If the intersection of two polyhedra is not empty, then there exists at least one pair of iterations that is executed in the wrong order, thus rendering the transformation *illegal*. The whole process is called *Violated Dependence Analysis* [38].

The task of GRAPHITE is to look for such transformations that are beneficial for optimizing various criteria, but which are legal at the same time. The simple search heuristic is looking for good transformations, rejecting those which are illegal. This is certainly an iterative process. If none of the transformations seems legal, then no transformation is done. GRAPHITE currently implements loop interchange, loop strip-mining, loop distribution and loop-blocking.

3.6 Code generation

In source-to-source polyhedral compilers, the code generation pass is the last one, generating the new loop structures to scan statement instances in the order defined by the modified schedule.

In GRAPHITE it is not the syntactical source code that is the final result of the pass: GRAPHITE should be able to regenerate the GIMPLE code. Furthermore, the generated GIMPLE code has to be reinserted back into the CFG, respecting the SSA form invariants and passed to the further passes after GRAPHITE.

Multiple loop generation tools exist that operate on the polyhedral model. The most mature one is the CLooG (Chunky LOOp Generator) [4]. CLooG is used in GRAPHITE as the major component of code generation. Since CLooG is meant for generating syntactic code (mainly `C` code), it cannot be used directly: CLooG generates an internal representation called CLAST which is a simple abstract-syntax tree containing only loops, conditions, and statements. In our case statements are replaced with basic blocks.

CLooG is fed by the polyhedral representation (GPOLY) and is asked to generate a CLAST. The nodes of the abstract-syntax tree are pointers to original basic blocks. Depending on the loop transformations, the basic blocks might be rescheduled, moved

to other loops, or even replicated (when performing a transformation). The final effect is represented in the CLAST. The CLAST tree is traversed and the basic blocks are put into the their new positions in the GIMPLE CFG, loop structures are regenerated and some basic blocks are replicated.

Even in the case of the identity transformation (no schedule modification), the newly generated loops according to the CLAST tree have the new induction variables. All the basic blocks belonging to a SCoP have to be scanned, and the old induction variables have to be replaced with new induction variables.

3.7 Algorithm choices and compilation speed

Polyhedral optimizers have challenges with scalability and GRAPHITE is no exception. While developing GRAPHITE, we have encountered some interesting issues that affect compilation speed and are exploring algorithms choices to improve performance.

One example is loop unrolling. C++ templates are a powerful feature used in many high-performance codes; template meta-programming is combined with inlining to produce specialized loops. This style creates a large abstraction penalty that GCC has chosen to address with an early inner loop unrolling pass. Applications (such as Tramp3D in the GCC test suite) show significant performance improvement through this technique. However, this also affects loop and data dependence analysis for optimizations such as auto-vectorization and GRAPHITE, whose analysis and compilation time grows with the number of variables in each SCoP. We explore ways to tune this unrolling in the presence of GRAPHITE and eventually to implement such unrolling within GRAPHITE itself.

4 Optimizations

Loops that carry no dependence may be good candidates to be parallelized, i.e, different iterations of the loop might be executed simultaneously by multiple threads [1]. In GCC, two main infrastructures are used to accomplish loop parallelization: data dependency analysis and the GNU OpenMP library. OpenMP defines language extensions to C, C++, and Fortran for implementing multithreaded shared memory applications [29]. Automatic generation of such extensions by the compiler relieves programmers from the manual parallelization process. OpenMP support has been implemented in GCC since version 4.2 [27], and together with existing data dependence analyses, opened the door for automatic parallelization in GCC.

Automatic parallelization was first implemented in version 4.3 as a technology preview. It is able to detect loops carrying no dependences, and generate parallel code by creating and inserting the necessary OpenMP structures and support. It is triggered by `-ftree-parallelize-loops=`x, where x defines the number of threads to create.

Generating parallel code Once the GCC auto-parallelizer decides to parallelize a loop, it generates the parallel code using the OpenMP structures to define the parallel section and relevant attributes like the scheduling method, the shared vs. private variables, atomic operations etc.

In Figure 9, we see an example of a sequential loop, and the parallel code generated for it (assuming the number of threads requested by the user is 4). `.paral_data` is a structure field gathering all the shared data that should be provided for each

thread. The loop is outlined to a separate function, `parloop._loopfn()`, which is run individually by each thread, supplied with the shared data.

The GNU OpenMP library provides two builtins which define the parallel section: `GOMP_parallel_start()` creates the threads. `GOMP_parallel_end()` is a barrier where all the threads are joined. After `GOMP_parallel_start()` is executed, 4 threads are created. Each thread is executing the outlined function, iterating the loop with a different (and exclusive) interval of iterations, represented as start and end at the example. After `GOMP_parallel_end()` is executed, the threads are joined back to one master thread.

```
parloop
{
  for (i = 0; i < N; i++)
    x[i] = i + 3;
}
```

(a)

```
parloop
{
  .paral_data.x = &x;
  __builtin_GOMP_parallel_start (parloop._loopfn,
                                 &.paral_data, 4);
  parloop._loopfn (&.paral_data);

  __builtin_GOMP_parallel_end ();
}

parloop._loopfn (.paral_data)
{
  for (i = start; i < end; i++)
    (*.paral_data->x)[i] = i + 3;
}
```

(b)

Fig. 9. (a) sequential loop (b) parallel code generated

```
for i
  for j
    A[i][j] = A[i-1][j]
```

(a)

```
for j
  for i
    A[i][j] = A[i-1][j]
```

(b)

Fig. 10. (a) the original loop (b) after interchange

Integration of the parallelizer with graphite The initial analysis used for the parallelizer was based on the Lambda framework [24]. It has been replaced with the GRAPHITE based dependence analysis.

Integrating the parallelizer with GRAPHITE is profitable for a number of reasons:

- GRAPHITE dependence analysis is more accurate than Lambda, hence could detect more parallel loops [38].
- The ability of GRAPHITE to perform long and complex compositions of program transformations enables to extract more parallelism [13] and to optimize for parallelism and locality simultaneously [7].
- Since GRAPHITE is able to represent sequences of loop transformations as a single scheduling transformation, it seems natural to incorporate a cost model into it to control the transformation sequence. Parallelization is a key transformation whose cost and benefit should be applied to such a model, in the hope of deriving the

most profitable combination of loop transformations. We worked on such a cost model in the special case of automatic vectorization [37], extension to more general parallelization and to the management of temporal locality is in progress.

Figure 10 shows a simple example demonstrating the interaction of loop parallelization with another transformation, loop interchange. The original loop is shown in Figure 10(a). The outer loop carries a dependence and therefore can't be parallelized. Parallelizing the inner loop is possible, but results in executing a synchronization barrier at the end of each outer-loop iteration, therefore executing a synchronization `i` times. If, however, we interchange the loop, as shown in (b), we can parallelize the outer-loop, resulting in use of just one barrier.

Automatic parallelization was integrated to GRAPHITE as part of the upcoming GCC4.5. In addition to `-ftree-parallelize-loops=`x, `-floop-parallelize-all` is specified to enable it as a GRAPHITE-based transformation.

5 Alias Information and Polyhedra

Alias analysis is an intrinsic module of any compiler as it facilitates any other optimization that involves variable disambiguation, such as scheduling or identifying invariants, redundant subexpressions, etc. For scalability reasons, most compilers use fast but rather imprecise analysis like Anderson's algorithm [2], a context-insensitive, flow-insensitive subset-based may-alias analysis. GCC relies on an extension of this algorithm that is field-sensitive as well [5, 30].

A data-reference is either a scalar variable, or an array-reference, or an offset of an array by a compile-time constant, or an offset of an array by an index, or a pointer variable. The difference between the latter four types in `C` is that the first three resolve to a constant pointer (like `const int *`) referring to a stack location, while the last one can only be resolved to an ordinary pointer (like `int *`) referring to a heap location.

Example In the following code excerpt, it can be seen that a and p may-alias to each other, and so do p and b, but a and b do not.

```
int a[10], b[10];
void foo (int *p);
```

Most alias analysis algorithms return a points-to relation, where data references are mapped to abstract stack or heap locations called *alias sets*. We will also refer to this relation as the forward mapping. On the above example it is: $a \rightarrow \{A_1\}$, $p \rightarrow \{A_1, A_2\}$, $b \rightarrow \{A_2\}$.

In GCC (since version 4.4), the result of the alias-analysis is encoded in an *alias oracle* that returns the information about *presence or absence* of may-alias relation between pairs of data references. It can be seen that such a portable interface goes well with various scalar analyses that use it.

The information that is provided by alias-oracle can be represented as an undirected graph, whose vertices correspond to data-references and whose edges represent presence of alias-relationship between pairs of data-references.

The aliasing relation for the previous example can be represented as a graph

$$G_a\colon a \text{ — } p \text{ — } b.$$

It is known that polyhedral dependence analysis for a given SCoP usually makes $\mathcal{O}(n^2)$ polyhedral operations, when n is the number of convex polyhedra representing memory references. Dependence analysis can exploit the properties of G_a such that the number of calls to polyhedral libraries can be reduced. The above examples show

that the dependence analysis could effectively use the information provided by the alias-analysis to its full potential.

To represent the alias information, GRAPHITE creates an additional dimension (the first) for each array reference. This additional dimension, henceforth called *alias dimension* is indexed by the alias set to which that particular data reference points to. A data reference however, could be a member of more than one alias set. For example, variable p in the above example, belongs to two alias-sets A_1 and A_2. Though this may be because of impreciseness of the algorithm used in the alias analysis, it could as well be because of a true aliasing of the associated memory regions. Hence, GRAPHITE indexes the alias dimension by the disjunction of alias-sets to which that particular data-reference points to.

In the above example, if we let $\mathcal{D}_a^R$, $\mathcal{D}_p^R$ and $\mathcal{D}_b^R$ be the original polyhedral domains of the three variables a, p and b, then the corresponding memory references, annotated by the alias dimension would be `memory_reference`$[A_1, \mathcal{D}_a^R]$, `memory_reference`$[A_1 \vee A_2, \mathcal{D}_p^R]$, and `memory_reference`$[A_2, \mathcal{D}_b^R]$ respectively.

Definition: Minimum Edge Clique Cover For an undirected graph $G = (V, E)$, the Minimum-Edge Clique Cover is defined to be a collection $A_1, A_2, \ldots, A_k$ of subsets of V, such that each A_i induces a complete subgraph of G and such that for each edge $(u, v) \in E$, there is some A_i that contains both u and v. This problem is also called *Edge Clique Cover* (ECC). Another problem similar to ECC is the *Vertex Clique Cover* (VCC) that computes a cover on cliques of vertices.

It is easy to see that if there is a clique in G_a, then polyhedral dependence analysis should test for dependence between all variables participating in the clique to determine the nature of dependence between the data-references participating in the clique. If on the other hand, there is no edge between a pair of variables, there is no need for polyhedral operations. This information is equivalent to *cliques* in G_a. In the above example, it can be seen that there are two 2-cliques: $\{a, p\}$ and $\{p, b\}$. Polyhedral dependence analysis should test for dependence between each of these pairs. It however does not need to test for dependence between a and b.

It is clear that maximizing the clique size in G_a is helpful. But, it is the edges of G_a that correspond to possible intersections of memory areas, thereby corresponding to aliasing of data-references. Hence, the problem that minimizes the number of representative elements in the alias-dimension should be an edge-clique, thereby meaning ECC (rather than vertex-clique or maximum clique). Further, as the edges are *covered* by the representative element, along with all possible edges which could alias with it, the dependence analysis can thus use alias-set representatives to drive its algorithm.

5.1 ECC: problem and solution

The ECC problem is an NP-Hard problem (page 194 in [12]). It is different from the more widely studied VCC problem, which is closely related to the graph coloring problem. The VCC solution for a graph G, though being a NP-Hard problem itself – could be trivially found after coloring the complement graph G'.

No fast running and close to optimal heuristic is known for ECC. It may be very hard to solve even in an approximate sense. On the other hand, the other above mentioned problems (VCC and graph coloring) have linear-time and exact solutions for special classes [14].

The algorithms for ECC either solve exactly by a brute-force search, or by using a heuristic developed for VCC or graph-coloring. In Gramm et. al's paper [15], which

is state of the art for this problem, new algorithms for both methods are suggested. In their paper, the running time of a previously known heuristic is improved from $\mathcal{O}(|V||E|^2)$ to a more acceptable $\mathcal{O}(|V||E|)$. The major contribution of the paper however, are data-reduction rules that help reduce the input problem size by preprocessing. The method suggested in [15] is that the resultant graph after iteratively applying the rules is usually smaller, and hence can be subjected to either of the above mentioned methods (exact-solution or heuristic) for a faster solution.

5.2 Empirical analysis of alias graphs

We have done an empirical analysis of the graphs that are returned by the alias-analysis currently in GCC.

Of the 4481 graphs from SPEC Cpu 2006 benchmarks, 4367 are trivial, with the definition of trivial being $|V| < 10 \vee |E| < 5$. Only 328 graphs are non-trivial. Of the latter kind of graphs, only 11 graphs are interesting. In the rest of the graphs, every connected component is a clique. In all the graphs, the number of vertices participating in the maximal cliques vary in the range $1 \leq |V| \leq 90$.

From the above empirical analysis, one could say that the alias-oracle which marks every connected component as a clique is generally very imprecise and hence advanced algorithms for solving ECC are not needed in the present context. A general counter-example to such a reasoning is the large size of graphs, taken from real-world examples, containing a wide range of maximal cliques. Further, we also have specific counter-examples which show that exceptions to the above statement exist in real world. Figure 11 depicts the alias relation for a kernel in the GCC-testsuite extracted from an H.264 decoder. Each ellipse on the (i) graph represents a clique. A block-edge between two ellipses X and Y represents an edge between every pair of vertices in the set $\{X, Y\}$.

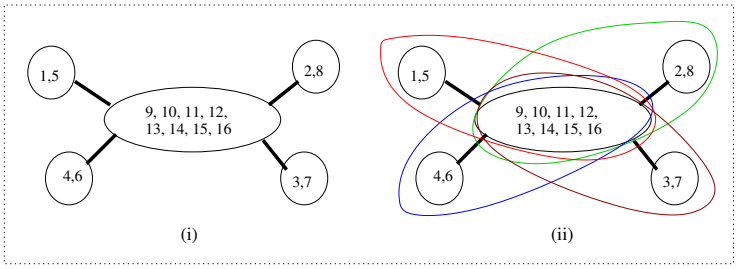


Fig. 11. Alias relation for an H.264 kernel.

The optimal solution of ECC is shown on the (ii) graph. It has 4 cliques, each of which is a union of one of the smaller cliques with the biggest clique.

We now describe the polynomial algorithm that we chose to compute an ECC:
Input: an alias graph G_a
Output: a mapping from vertices to the alias-set representatives.

- Do a Depth First Search (DFS) on G_a and separate out the connected components $A_1, A_2, \ldots, A_k$. This step takes $\max(|V(G_a)|, |E(G_a)|)$ time.
- Check if each connected component A_i is a clique or not. This step takes $|V(A_i)|^2$ time, where $V(A_i)$ is the number of nodes of the connected component A_i.

– If A_i is small enough ($|V| < 5 \vee |E| < 10$), then search for solution using a direct search.
– Apply the simple and cheaper data-reduction rules (mainly Rule 1 and Rule 2), as explained in [15], so that the problem size could be reduced.

Currently, we are using DFS based numbering, testing for cliques, and simple search. Though this method is very conservative, it gives the optimal solution for most of the cases for SPEC Cpu 2006, though not for the H.264 example shown above. For this example, this solution returns that all edges are in the same cover. Thus our current method is not searching for a clique, which leads to loss of precision, and needs to be improved. We are working on another heuristic approach to design a polyhedral algorithm computing a suboptimal ECC but without loss of points-to information.

6 Conclusion

We presented the design of the GRAPHITE pass of GCC, focusing on the challenges and novel research issues arising from this confrontation of polyhedral compilation with the real world. Our work makes the following contributions:

– We implemented the polyhedral model on a three-address, SSA-based representation, opening interesting reuse and interaction opportunities for analyses and optimizations in production compilers.
– We extended the polyhedral representation to capture alias relations among pointer-based data references, with no impact on polyhedral dependence analysis and transformation algorithms.
– We also extended this representation to capture scalar dependences and reductions.
– We set the framework for aggregating statements into "polyhedral basic blocks" or splitting those blocks into smaller components, with the ability to trade expressiveness for compilation time.
– We motivated further research on the practical interaction between polyhedral loop transformations and other optimizations, including parallelization and vectorization.

6.1 Prospective work

There are two main issues that are the focus of the prospective work on automatic parallelization:

– **Heuristics/cost model for automatic parallelization**. Currently, a very simple method is used to determine whether it is profitable to parallelize a certain loop. We need a good model to determine if we should parallelize a loop considering performance reasons.
– **Advanced automatic parallelization**. Currently we are only able to detect whether a loop is parallel or not. We would like to explicitly apply transformations to increase and expose further parallelism opportunities, and we have shown that GRAPHITE is the right setting to design such transformations.

Acknowledgments. Tobias Grosser was supported by AMD as a summer intern and by a Google Summer of Code grant. Konrad Trifunovic was supported by IBM and the HiPEAC FP7 European network as a summer intern. Li Feng was supported by a Google Summer of Code grant. GRAPHITE was partially supported by the ACOTES FP6 European project.

References

1. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan and Kaufman, 2002.
2. L. O. Andersen. Program analysis and specialization for the c programming language. Technical report, DIKU, 1994.
3. R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.
4. C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Parallel Architectures and Compilation Techniques (PACT'04)*, Antibes, France, Sept. 2004.
5. D. Berlin. Structure aliasing in GCC. In *the GCC Developers' Summit*, pages 25–36, 2005. http://www.gccsummit.org/2005.
6. W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, Dec. 1996.
7. U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelization and locality optimization system. In *ACM SIGPLAN Conf. on Programming Languages Design and Implementation (PLDI'08)*, Tucson, AZ, USA, June 2008.
8. C. Chen, J. Chame, and M. Hall. CHiLL: A framework for composing high-level loop transformations. Technical Report 08-897, U. of Southern California, 2008.
9. A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, and N. Vasilache. Facilitating the search for compositions of program transformations. In *ACM International conference on Supercomputing*, pages 151–160, June 2005.
10. K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, 1993.
11. P. Feautrier. Some efficient solutions to the affine scheduling problem, part II, multidimensional time. *Intl. J. of Parallel Programming*, 21(6):389–420, Dec. 1992. See also Part I, one dimensional time, 21(5):315–348.
12. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
13. S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Intl. J. of Parallel Programming*, 34(3):261–317, June 2006. Special issue on Microgrids.
14. M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Elsevier, 2nd edition, 2004.
15. J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Data reduction and exact algorithms for clique cover. *J. Exp. Algorithmics*, 13:2.2–2.15, 2009.
16. M. Hall et al. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, Dec. 1996.
17. F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the pips project. In *ACM Intl. Conf. on Supercomputing (ICS'91)*, Cologne, Germany, June 1991.
18. R. Johnson, D. Pearson, and K. Pingali. The program structure tree: computing control regions in linear time. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 171–185, New York, NY, USA, 1994. ACM.
19. KAP C/OpenMP for Tru64 UNIX and KAP DEC Fortran for Digital UNIX. http://www.hp.com/techsevers/software/kap.html.
20. W. Kelly. Optimization within a unified transformation framework. Technical Report CS-TR-3725, University of Maryland, 1996.

21. W. Kelly. Optimization within a unified transformation framework. Technical Report CS-TR-3725, Department of Computer Science, University of Maryland at College Park, 1996.
22. W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-3193, University of Maryland, 1993.
23. W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers'95 Symp. on the frontiers of massively parallel computation*, McLean, 1995.
24. W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *Intl. J. of Parallel Programming*, 22(2):183–205, April 1994.
25. The LooPo Project - Loop parallelization in the polytope model. http://www.fmi.uni-passau.de/loopo.
26. B. Meister, A. Leung, N. Vasilache, D. Wohlford, C. Bastoul, and R. Lethin. Productivity via automatic code generation for pgas platforms with the r-stream compiler. In *APGAS'09 Workshop on Asynchrony in the PGAS Programming Model*, Yorktown Heights, New York, June 2009.
27. D. Novillo. Openmp and automatic parallelization in gcc. In *the GCC Developer's summit*, June 2006.
28. M. O'Boyle. MARS: a distributed memory approach to shared memory compilation. In *Proc. Language, Compilers and Runtime Systems for Scalable Computing*, Pittsburgh, May 1998. Springer-Verlag.
29. The OpenMP API specification for parallel programming. http://openmp.org/wp/.
30. D. J. Pearce, P. H. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis of C. *ACM Trans. Program. Lang. Syst.*, 30(1):4, 2007.
31. PLUTO: A polyhedral automatic parallelizer and locality optimizer for multicores. http://pluto-compiler.sourceforge.net.
32. S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber, and N. Vasilache. Graphite: Loop optimizations based on the polyhedral model for GCC. In *Proc. of the 4^{th} GCC Developer's Summit*, Ottawa, Canada, June 2006.
33. S. Pop, A. Cohen, and G.-A. Silber. Induction variable analysis with delayed abstractions. In *Intl. Conf. on High Performance Embedded Architectures and Compilers (HiPEAC'05)*, number 3793 in LNCS, pages 218–232, Barcelona, Spain, Nov. 2005. Springer-Verlag.
34. L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM Conf. on Programming Language Design and Implementation (PLDI'08)*, Tucson, Arizona, June 2008.
35. L. Renganarayana, U. Bondhugula, S. Derisavi, A. E. Eichenberger, and K. O'Brien. Compact multi-dimensional kernel extraction for register tiling. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.
36. A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable autotuning framework for computer optimization. In *IPDPS'09*, Rome, May 2009.
37. K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *Parallel Architectures and Compilation Techniques (PACT'09)*, Raleigh, North Carolina, Sept. 2009.
38. N. Vasilache, A. Cohen, C. Bastoul, and S. Girbal. Violated dependence analysis. In *ACM Intl. Conf. on Supercomputing (ICS'06)*, Cairns, Australia, June 2006.
39. N. Vasilache, A. Cohen, and L.-N. Pouchet. Automatic correction of loop transformations. In *Parallel Architectures and Compilation Techniques (PACT'07)*, Brasov, Romania, Sept. 2007.

Extending GCC with a multi-grain parallelism adaptation framework for MPSoCs

Nicolas BENOIT and Stéphane LOUISE

CEA LIST, Embedded Real Time Systems Laboratory,
Point Courrier 94, Gif-sur-Yvette, F-91191 France
{firstname.lastname}@cea.fr

Abstract. Multiprocessor-System-on-a-Chip architectures offer multiple granularities of parallelism. While harnessing the lowest levels by means of vectorization and instruction scheduling only requires local information about the code and one of the cores, coarser levels raise inter-dependent trade-offs which necessitate a global approach.
This paper introduces Gomet: an extension to GCC which combines a hierarchical intermediate representation of the input program and a high-level machine description to achieve multi-grain parallelism adaptation. Gomet builds this representation from an inter-procedural dependence analysis, and transforms it to fit the target hardware. Then, it generates specialized C source files to feed the native compiler of each core of the target. Early evaluation of Gomet with simple programs show encouraging results and motivate further developments.

1 Introduction

On-going research on the programming of emerging massively parallel architectures [1–4] has brought new languages, new programming models and revived interests in automatic parallelization techniques [5–7]. However, by decoupling parallelism expression from the target architecture, a new abstraction gap between the software and the hardware has been created. To succeed in the simplification of parallel programming and achieve portability, compilation tools (with the possible help of runtime libraries) must bridge that gap. In other words, the high-level parallelism abstraction must be adapted (or lowered) to the hardware features available.

Multiprocessor-System-on-a-Chip architectures (MPSoCs) support multiple levels of parallelism and offer specialized processing units [1, 3]. Consequently, the possible mappings of a program to a given architecture vary in different performance metrics: execution time, memory footprint, communication volume, energy consumption. At the finest levels of parallelism (SIMD, MIMD), the inter-dependent trade-offs raised can be solved locally, and are supported by most compilers. On the other side, the efficient handling of coarser granularities requires in the compilation flow a confrontation between the whole program and the target architecture. Figure 1 provides an illustration of such holistic parallelism adaptation in the compilation flow.

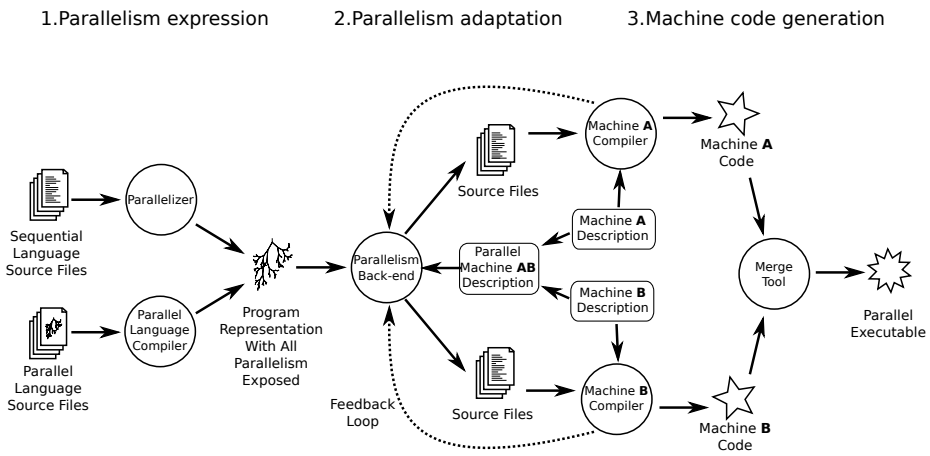


Fig. 1. Compilation flow integrating parallelism adaptation.

The first step consists in capturing as much parallelism as possible in the input program. This can be achieved either with an automatic parallelization tool or by providing specific language constructs to the programmer. During the second step, a parallelism back-end adapts the exposed parallelism to the physical execution resources of the target architecture. Finally, for each type of execution resource, a set of separate source files is generated and processed with the native compilation tools (step 3). In an iterative compilation scheme, the latter can guide the parallelism back-end by providing feedbacks.

Our research work investigates the coupling of the parallelism back-end with native compilers for MPSoCs. Among others, we are interested in the support of hardware mechanisms such as weak synchronization [8] between processing units, and the automatic accounting of heterogeneous resources.

To experiment with parallelism adaptation, we are developing a new extension to GCC called Gomet. Its front-end builds a hierarchical intermediate representation of the input program. Its back-end confronts the obtained representation with a high-level machine description and generates specialized parallel C source files.

This paper describes the general architecture of Gomet and is organized as follow: section 2 presents the related work, section 3 details the front-end internals, section 4 presents the parallelism back-end code generator, and section 5 shows the results obtained as the development reaches its first milestone.

2 Related Work

Harnessing the massive parallelism offered by current and forthcoming architectures is a long-term and very active field of research. Among the numerous propositions to enhance the support of parallelism in the compilation flow [9–14], this section presents the work which share the most similarities with Gomet.

2.1 Outside GCC

The OSCAR Compiler [9] integrates a multi-grain parallelizer based on the macro-dataflow [15] program representation. The macro-dataflow is a directed hierarchical graph which describes macro-tasks and their data dependencies at various levels of granularity: subroutine calls, repetitive blocks and statements. The compiler schedules the graph on the target architecture and uses a high level API [16] to abstract hardware mechanisms. This API comprises energy consumption controls and a subset of OpenMP [17]. After its translation by a target dedicated back-end, the native compiler can produce the final executable.

The ACOTES [10] research project aims at providing programmers with a stream-oriented programming environment. It defines a set of directives extending OpenMP in order to abstract stream management. Those directives are processed with the Mercurium source-to-source compiler [18] which replaces directives with calls to a dedicated library. It relies on a high-level machine description and a simulator to schedule the streams on the available resources and operate transformations such as task fusion. The output of the source-to-source compiler is processed by GCC in order to produce the final program.

Sage++ [19] is used to pre-process extended C++ with data-parallel constructs (pC++). It generates standard C++ code later compiled by the native compiler and linked to a machine-specific runtime system.

The Stream Virtual Machine [20] is a high-level machine description which captures the main features of stream processors. It comprises a machine model and a high-level API, which abstract computation and data partitioning, communication and synchronization.

The reader may find case studies of adaptation of stream programs to existing architectures in [21–23].

2.2 Inside GCC

Current versions of GCC support OpenMP 3.0 directives through *libgomp*. It is notably exploited by the automatic parallelization pass *autopar* which adds appropriate Gimple OMP statements (and cost estimation) each time the loop analysis detects a parallel loop. Towards supporting the pipeline parallel construct, Pop et al. recently presented automatic streamization [24].

Gomet experiments the integration of a hierarchical program representation into GCC and the generation of parallel C source code from it. This hierarchical representation allows the adaptation of the program at multiple levels of granularity. The transformation is driven by a generic process tuned by target-specific cost-models and program transformations.

3 Generation of a Hierarchical Intermediate Representation

This section introduces Gomet's front-end and how it generates a hierarchical intermediate representation of the input program, referred as Kimble, and cap-

tures as much information as possible about the available parallelism. Figure 2 shows where it is inserted in GCC flow.

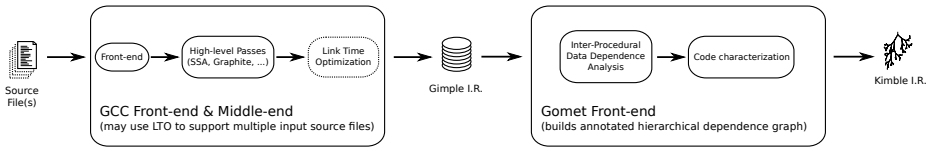


Fig. 2. Insertion of Gomet's front-end into GCC flow.

Currently, Gomet is implemented as one of the *-O1* optimization passes of *passes.c*. It requires the *-fgomet* command line switch to be activated. For each function present in a source file, the optimization passes sequence is interrupted before lowering Gimple I.R. to RTL. When the last function definition is reached, Gomet enters its processing chain. Among other high-level GCC optimization passes, Gomet benefits from the Static-Single-Assignment (SSA) form and Graphite loop transformations. The first one simplifies scalar data dependences by preventing multiple assignments of the same variable. The second one reduces the number of loop-carried dependences. Therefore, Gomet assumes that parallelism at the statement and loop levels is exposed and requires no further transformations. In order to cover coarser granularities of parallelism, it integrates an inter-procedural data-dependence analysis.

3.1 Inter-Procedural Data-Dependence Analysis

To push parallelism adaptation to its limits, the data-dependence analysis aims at capturing all the available parallelism in the input program. This is achieved by collecting exhaustively the memory references triggered during an emulation of its execution.

To be correct, the analysis assumes that the program's call graph contains no cycle and that all loop bounds and data are defined. Undefined functions are tolerated to the condition they use no data outside their own scope, this holds for arithmetic functions such as *sin()*, *sqrt()*, etc. Though restrictive, those assumptions are acceptable for this prototyping work. Moreover, they fit in the deterministic parallelism context of embedded systems which our research addresses.

Memory Access Formula. Prior to the analysis, a program parsing pass builds a virtual memory address space and associates a unique address to each data declared. Then, for each data access, a formula that will allow to compute the corresponding memory address can be established. The formulas are symbolic and contain unknown values at the time of their building. Currently, formulas support referencing, dereferencing, array indexing and composition (for example

structure fields accesses). Each formula is coupled with the number of bytes the access reads or writes.

Program Tree. The analysis abstracts the program as a tree, which can contain five types of nodes: function call, loop, iteration, basic-block and statement. The root of the tree is the root of the call-graph (for example the *main()* function), it corresponds to the coarsest level of granularity. The leaves are statements, which represent the finest level of granularity.

Processing. The analysis processes the tree depth-first, following the path of the execution flow of the program. In other words, it reaches the finest level of granularity before processing coarser levels. It emulates the behavior of function calls, loops and simple arithmetic statements, as if the code was partially executed. When the statement level of a branch is reached, memory accesses formulas are evaluated. Each time the analysis finishes the processing of the children of a node, it builds a summary of their memory accesses and a dependence graph between them. A memory accesses summary classifies accessed memory addresses into one of the three following sets: Read-Only, Read-Write, Write-First [25].

Current Implementation. The proposed analysis exhaustively computes all memory references in the input program and tests if they intersect. It is a simple and reliable approach, which captures all exposed parallelism. Nevertheless, its duration depends on the input code, requiring hours of analysis for large iteration domains. For example, the initialization and the product of two 256×256 matrices takes about 20 minutes on an Intel Xeon clocked at 3 GHz. On the other hand, memory usage is kept low by freeing memory accesses summaries as soon as they have been aggregated at a coarser level of granularity. This may allow to duplicate the contexts of large loops in order to analyze multiple iterations in parallel.
In the future, the analysis could be hybridized with traditional dependency tests or polyhedral approaches to detach its complexity from the input code. Existing components of GCC (Graphite, OpenMP support, auto-vectorization) can help to achieve this goal.

3.2 Gimple Encapsulation with Kimble

In order to store the information collected during the inter-procedural analysis, the Gimple intermediate representation is encapsulated into Kimble, a hierarchical structure of dependence graphs. Kimble wraps Gimple at the statement level, and adds containers that map nested constructs with coarser granularities. Containers at the same level form a DAG where edges describe data-dependence relationships. Other examples of hierarchical dependence graphs can be found in [26–29, 15].

Our intermediate representation follows an organization similar to [15], but defines six types of nodes to remain closer to the level of the C source code that will be generated. Four of those node types come from the program tree built during the dependence analysis. Loops can be annotated with the type of parallelism they support: *undividable*, *map* (all iterations are independent) or *reduce* (iterations expose a reduction dependence scheme).

$$
\begin{aligned}
&Nodes = \{Function, Loop, Region, Cluster, Statement, Call\} \\
&\left.\begin{array}{rl} Tree_N \rightarrow & (Tree_N) \\ | & Tree_N \parallel Tree_N \\ | & Tree_N ; Tree_N \\ | & n \in N \end{array}\right\} \text{Dependence graph expression} \\
&\left.\begin{array}{ll} Function \rightarrow & Function(Tree_{\{Loop,Region\}}) \\ Loop \rightarrow & Loop(Tree_{\{Loop,Region\}}) \\ Region \rightarrow & Region(Tree_{\{Cluster,Statement,Call\}}) \\ Call \rightarrow & Call(Tree_{\{Function\}}) \end{array}\right\} \text{Hierarchy expression}
\end{aligned}
$$

Fig. 3. Grammar ruling Kimble structure.

Figure 3 gives an overview of the grammar ruling Kimble structure. Node types are given in the $Nodes$ set. Nodes at the same level are linked using $\parallel$ and ; operations which respectively establish parallel and sequential relationships and express dependence. Then, legal hierarchical (a.k.a. nested) constructs can be read as following : "A function contains a dependence graph of *Loop* and *Region* nodes". We may mention that *Region* and $Cluster$ nodes correspond respectively to a basic-block and an undividable (possibly empty) group of statements.

As the grammar ruling Kimble suggests, the parallelism information expressed directly map to C constructs. This eases parallelism adaptation as this information is conserved and transparently updated during program transformation.

3.3 Example

Kimble representation is illustrated with a function extracted from the x264 [30] H.264 video encoder: *sub16x16_dct()*. This function performs a DCT on the difference of two 16×16 matrices.

Figure 4 is a simplified Kimble representation of this function, the SSA form was reduced in order to limit the graph size. Arrows indicate a dependence relationship, while dotted edges represent a hierarchical link, also referred as nesting relationship.

Within *sub16x16_dct()*, the representation highlights the independence of four calls to *sub8x8_dct()*. Itself embeds four independent calls to *sub4x4_dct()*, which

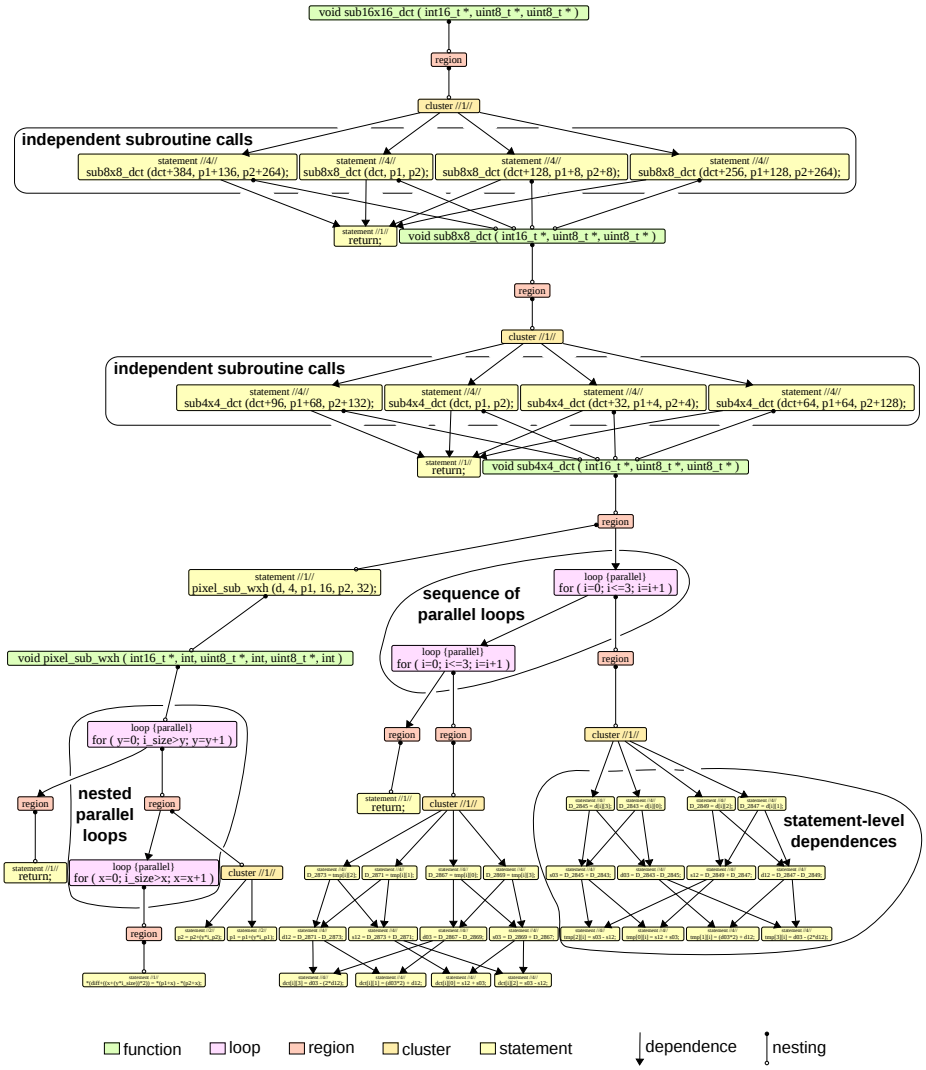


Fig. 4. Simplified Kimble representation of the sub16x16_dct() function in x264.

contains a reference to *pixel_sub_wxh()* and two successive loops. Between braces, the loops are tagged as being *parallel*. Between double slashes, the statements are annotated with their concurrency level metric, collected during code characterization.

3.4 Code Characterization

In order to hint the parallelism mapping decisions, a few static information are collected and decorate the nodes of the Kimble tree. It includes for example the number of integer operations, the volume of data written, etc.

Another metric used for code characterization is the concurrency level supported by each node. It corresponds to the minimum number of nodes that can be executed concurrently at the same hierarchy level. It is computed using the dependence graph at each level of hierarchy within the Kimble representation.

In the future, characterization should take advantage of the information already gathered and computed by GCC itself. It could also employ a communication interface with the native compiler or other static analysis tools, and exploit profiling data.

4 Program Transformation and Code Generation

This section describes the back-end of Gomet: how it adapts the parallelism exposed in the Kimble representation and how it outputs C source code. Figure 5 shows the flow in which it operates.

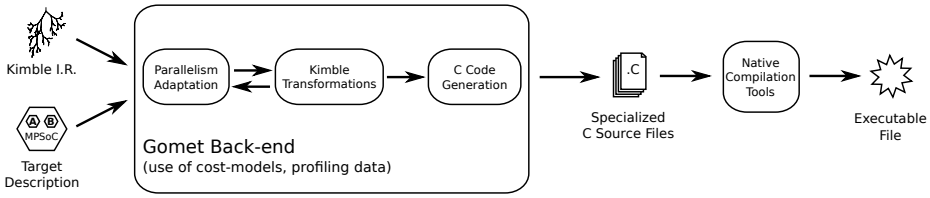


Fig. 5. Inputs and outputs of Gomet's back-end.

The Kimble representation is iteratively transformed to map parallel branches of the tree to the available resources of the architecture. When this process is done, the tree is walked to generate C source code.

4.1 Kimble Transformations

The Kimble representation can be simplified and modified by means of four transformations:

Pruning. The pruning transformation removes empty nodes which are notably added during the systematic construction of the representation.

Aggregation. The aggregation transformation encapsulates chains of dependent statements into clusters.

Encapsulation. Also known as outlining (opposed to inlining), the encapsulation transformation detaches a branch from its context and inserts it into a newly created function. The variables shared between the detached branch and its environment are either put into a structure or passed as the function parameters. Besides isolating parallel tasks, this transformation allows to factorize code, addressing MPSoCs constrainted environments.

Loop Fission. The loop fission transformation creates an outer loop which redefines the bounds of the transformed loop so that multiple sub-domains can be processed independently. Currently, this transformation requires all iterations to be independent (*map* parallelism). It allows SPMD (Single-Program Multiple-Data) parallelism.

4.2 Adaptation to the Target Architecture

The adaptation process consists in coupling, through a dedicated API, a generic tree traversal and transformation procedure with a target architecture description. The latter is selected when invoking GCC+Gomet with the command line switch *-fgomet-target*.

Target Architecture Description. A target architecture description implements a set of callbacks to be used by the generic tree traversal procedure. The first kind of callbacks implements cost models: computations, communications, energy, etc. They are fed with the code characteristics gathered at the time of the Kimble representation building. The second kind of callbacks implements Kimble modifiers for parallelism implementation. For example, it may transform and generate bits of Kimble to insert calls to dedicated fork/join intrinsics, vector instructions intrinsics, communication primitives, etc. In this perspective, the target description can reference external libraries. The last kind of callbacks concerns the state of the machine, for example it updates the number of remaining free execution units.

Tree Traversal and Transformation. The traversal begins at the coarsest level of granularity, i.e. the entry point of the program, and implements the node decomposition technique described in [29]. For each set of concurrent nodes, a set of cost models is used to determine if offloading them to one of the available execution resource would be beneficial. If not enough concurrency is exposed,

the algorithm considers the slicing of parallel loops. If there is still not enough concurrency, the tree traversal dives to consider a finer level of granularity. When execution resources are exhausted or the offloading is impossible, the traversal quits the current level of granularity and resumes its work on the next set of concurrent nodes at the higher level.

4.3 C Source Code Output

Once the adaptation process is finished, the Kimble tree of each function is walked in order to generate the corresponding C source code.

GCC and Gimple tree codes are associated to their C language idioms, and SSA temporary variables are conserved and declared appropriately. Then, when processing a C program, statements can be straightforwardly unparsed from their Kimble representation. Loops are encapsulated using the usual C construct *for*, while other control flow constructs are restored using *if* and *goto* statements.

Other input languages were not extensively tested, as the purpose of Gomet is not to become a language conversion tool. However, converting Fortran canonical types and loops seems to be sufficient to support a subset of that language.

Global variables, types and structures defined in the input program are restored by inserting their definition at the top of the generated file. If the target architecture uses a runtime library, for example Pthreads, appropriate headers are also included.

Figure 6 is a shortened sample of the code generated for the sub4x4_dct() function in x264. It shows the SSA form of statements and reconstituted loop constructs.

5 Experiments

In this section, the results of the processing of four simple programs with GCC+Gomet are presented. Though it targets MPSoCs, the first milestone of Gomet does not include a machine description for such architecture, the C source code generated uses a Pthreads execution model, as proof of concept.
Gomet was built within GCC trunk revision 153048, while the reference executable and the output of GCC+Gomet were compiled with GCC 4.3.2. The target machine is a quad-core Intel E5320 clocked at 1.86 GHz with 4 MB of L2 cache. It has 4 GB RAM and runs Debian GNU/Linux with a 2.6.26 kernel. In order to limit the duration of the data-dependence analysis, the problem size was reduced and restored by hand in the generated code. Moreover, the repetition of the workload was manually forced so that it was large enough for the OS scheduler to allocate a physical core. Figure 7 shows the speed-up achieved when targeting one, two and four cores. For comparison purpose, the figure also shows the speed-up obtained after the program has been manually parallelized with OpenMP pragmas.
Program *A* implements the *sub16x16_dct()* function presented in 3.3, Gomet parallelized the calls to the subroutine *sub8x8_dct()*. Program *B* initializes three

```
1  void sub4x4_dct ( int16_t *dct, uint8_t *p1, uint8_t *p2 )
2  {
3    int16_t D_2820;
4    int D_2821;
5    [...]  /* more local declarations, including SSA variables */
6    int16_t d[4][4];
7    int d12;
8
9    goto R_13;  /* control flow restitution with gotos and labels */
10   R_13:
11    d_1 = (int16_t *) &(d);
12    pixel_sub_wxh ( d_1, 4, p1, 16, p2, 32 );
13    goto L_2;
14   L_2:
15    for ( i=0; i<=3; i=i+1 )  /* reconstituted loop construct */
16      {
17        goto R_11;
18   R_11:
19        D_2820 = d[i][0];
20        D_2821 = (int) D_2820;
21        D_2822 = d[i][3];
22        D_2823 = (int) D_2822;
23        s03 = D_2823 + D_2821;
24        [...]  /* more SSA  statements */
25        tmp[3][i] = D_2843;
26        goto R_12;
27   R_12:
28        goto L_2_;
29   L_2_:
30      ;
31      }
32    goto R_14;
33   L_1:
34    for ( i=0; i<=3; i=i+1 )  /* reconstituted loop construct */
35      {
36        goto R_9;
37   R_9:
38        D_2844 = tmp[i][0];
39        [...]  /* more SSA  statements */
40        D_2854 = (int16_t *) ((void *)dct + D_2853);
41        D_2854[2] = D_2838;
42        goto R_10;
43   R_10:
44        goto L_1_;
45   L_1_:
46      ;
47      }
48    goto R_15;
49   R_15:
50    return;
51   R_14:
52    goto L_1;
53 }
```

Fig. 6. Shortened sample of code generated by GCC+Gomet.

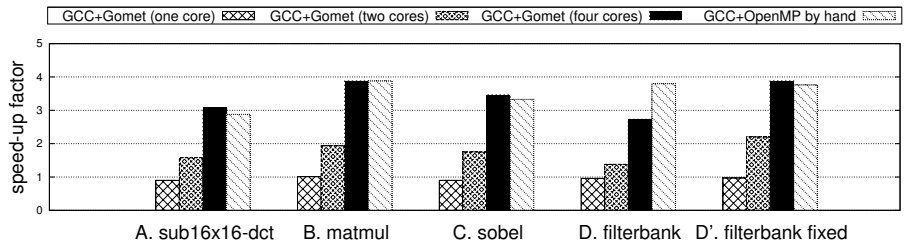


Fig. 7. Speed-up measurement of four programs processed with GCC+Gomet.

matrices A, B, C, and computes $C = C + AB$. The initialization of each matrix is performed in parallel and the outer loop of the matrix product is parallelized. Program C is a Sobel edge detection image filter. The horizontal and vertical gradients are computed in parallel, and on the 4 cores variant, their respective outer loops are parallelized. The outer loop of the magnitude computation is parallelized. Program D comes from the Stream-It [5] benchmark suite, it performs multi-rate signal processing. Gomet parallelized the outer loop of the processing to filter channels in parallel.
Those results show the ability of Gomet to generate valid parallel C source code, offering equivalent speed-ups to the ones obtained manually with OpenMP pragmas. However, they also suggest that the form of the code generated by Gomet affects the optimization passes of the native compiler. While little benefit can be measured for programs A and C, program D is negatively impacted. Figure 8

```
1   for ( k=0; (k<N_col)&(k<=j); ++k )
2     Vect_H[i][j] += H[i][k]*r[j-k];
```

A. Original code

```
1   Vect_H_I_I_lsm_2 = Vect_H[i][j];
2   D_2798 = 1;
3   for ( k=0; D_2798!=0;  )
4    {
5      D_2785 = Vect_H_I_I_lsm_2;
6      D_2788 = D_2787[k];
7      D_2789 = j - k;
8      D_2790 = (long unsigned int) D_2789;
9      D_2791 = D_2790 * sizeof(double);
10     D_2792 = (double *) ((void *)r + D_2791);
11     D_2793 = *(D_2792);
12     D_2794 = D_2788 * D_2793;
13     D_2795 = D_2785 + D_2794;
14     Vect_H_I_I_lsm_2 = D_2795;
15     k = k + 1;
16     D_2796 = k < N_col;
17     D_2797 = k <= j;
18     D_2798 = D_2796 && D_2797;
19   }
20  Vect_H[i][j] = Vect_H_I_I_lsm_2;
```

B. GCC+Gomet generated code

Fig. 8. Inner loop of a convolution code in *Filterbank*.

compares the original code of a convolution in the D program to the generated one, it shows two problems met with GCC 4.3.2 as the native compiler. First,

the exit condition of the loop is computed in the loop body (lines 16, 17, 18) and stored into a SSA temporary variable, making it difficult for GCC 4.3.2 to optimize the conditional jumps sequence. Second, the `Vect_H_I_I_lsm_2` value is duplicated into a SSA temporary variable (line 5), preventing GCC 4.3.2 from optimizing the accumulation (lines 13 and 14).
As figure 7 shows, the generated code for program D performs as well as the OpenMP code if manually fixed (by removing the mentioned SSA variables). In a future version of the code generator, an assignment chain compaction pass may be experimented to address this issue and explore the interaction between Gomet and the native compiler's optimizations.

6 Conclusion and Future Work

The mapping of parallel computations to MPSoCs raises complex inter-dependent trade-offs which require automated code generation tools. This paper introduced Gomet, an extension to GCC which enables the generation of parallelized C source code based on a hierarchical intermediate representation of the input program. This representation can express multiple granularities of parallelism, and is transformed to exploit the target architecture resources. Early experiments show encouraging results and validate the code generation approach used in Gomet.

The first milestone of the project intended to set up an effective parallelism adapation chain. The next milestone will focus on the support of more complex machine descriptions, distributed memory architectures and load balancing.

Besides, there are many other directions for future develoments in Gomet. First, the exhaustive dependence analysis could be hybridized with faster approaches when the code exposes regular patterns such as linear iteration spaces. Second, the machine description could interface Gomet with the native compilers of targeted cores, taking advantage of their detailed knowledge of instruction sets. Third, supporting the Link Time Optimization of the forthcoming GCC 4.5 would enable Gomet to deal with multiple input source files. Finally, in addition to C, the support of additional input languages and their features could be investigated.

7 Acknowledgements

The authors would like to thank Professor William Jalby for its insightful comments about this work.

References

1. Gschwind, M. et al.: Synergistic Processing in Cell's Multicore Architecture. IEEE MICRO **26**(2) (2006)
2. Wentzlaff, D. et al.: On-Chip Interconnection Architecture of the Tile Processor. IEEE Micro **27**(5) (2007)

3. Duller, A., Panesar, G., Towner, D.: Parallel Processing: the picoChip Way. Communicating Process Architectures (2003)
4. Greiner, A.: Tsar : a scalable, shared memory, many-cores architecture with global cache coherence. In: 9th Int. Forum on Embedded MPSoC and Multicore. (2009)
5. Thies, W., Karczmarek, M., Amarasinghe, S.P.: StreamIt: A Language for Streaming Applications. In: Computational Complexity. (2002)
6. Khronos OpenCL Working Group: The OpenCL Specification (Version 1.0) (2008)
7. Rus, S., Rauchwerger, L., Hoeflinger, J.: Hybrid analysis: static & dynamic memory reference analysis. International Journal of Parallel Programming **31**(4) (2003)
8. Calcado, F., Louise, S., David, V., Merigot, A.: Efficient use of processing cores on heterogeneous multicore architecture. In: CISIS '09. (2009)
9. Kimura, K. et al.: Multigrain Parallel Processing on Compiler Cooperative Chip Multiprocessor. (2005)
10. ACOTES: Advanced Compiler Technologies for Embedded Streaming: http://www.hitech-projects.com/euprojects/acotes/
11. Blume, W. et al.: Parallel Programming with Polaris. Computer **29**(12) (1996)
12. Polychronopoulos, C. et al.: Parafrase-2: an environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. International Journal of High Speed Computing **1**(1) (1989)
13. Irigoin, F., Triolet, R.: Semantical interprocedural parallelization: An overview of the PIPS project. In: ICS '91, ACM New York, NY, USA (1991)
14. Hall, Mary W. et al. : Interprocedural parallelization analysis in SUIF. ACM TOPLAS **27**(4) (2005)
15. Okamoto, M. et al.: Hierarchical macro-dataflow computation scheme. In: IEEE PACRIM '95. (1995)
16. Miyamoto, T. et al.: Parallelization with Automatic Parallelizing Compiler Generating Consumer Electronics Multicore API. In: IEEE APDCT '08. (2008)
17. OpenMP Architectural Review Board: OpenMP 3.0 specification (2008)
18. The Mercurium compiler: http://nanos.ac.upc.edu/content/mercurium-compiler
19. Bodin F. et al.: Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools. In: OONSKI '94. (1994)
20. Labonte, F. et al.: The stream virtual machine. In: PACT '04, IEEE (2004)
21. Gordon, M., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. ASPLOS '06 (2006) 151–162
22. Kudlur, M., Mahlke, S.: Orchestrating the execution of stream programs on multicore platforms. PLDI '08 (2008)
23. Udupa, A., Govindarajan, R., Thazhuthaveetil, M.J.: Software Pipelined Execution of Stream Programs on GPUs. In: CGO '09, IEEE Computer Society (2009)
24. Pop, A., Pop, S., Sjödin, J.: Automatic Streamization in GCC. In: 2009 GCC Developer's Summit. (2009)
25. Hoeflinger, J.: Interprocedural Parallelization Using Memory Classification Analysis. PhD thesis, University of Illinois at Urbana-Champaign (1998)
26. Warren, J.: A hierarchical basis for reordering transformations. In: POPL '84, ACM (1984)
27. Sarkar, V., Hennessy, J.: Partitioning parallel programs for macro-dataflow. In: ACM LFP '86, ACM (1986)
28. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3) (1987)
29. Hoang, P., Rabaey, J.: Scheduling of DSP Programs onto Multiprocessors for Maximum Throughput. IEEE Transactions on Signal Processing **41**(6) (1993)
30. x264 - a free h264/avc encoder: http://www.videolan.org/developers/x264.html

A case study: optimizing GCC on ARM for performance of `libevas` rasterization library

Dmitry Melnik[1], Andrey Belevantsev[1], Dmitry Plotnikov[1], and Semun Lee[2]

[1] Institute for System Programming, Russian Academy of Sciences
{dm,abel,dplotnikov}@ispras.ru
[2] Samsung Corp.
semun.lee@samsung.com

Abstract. This paper reports on the work for optimizing GCC on ARM to improve performance of `libevas` rasterization library. We used manual profiling and analysis as well as ACOVEA [3] compiler options tuning tool to identify weak places and tune GCC optimization parameters. We identified a number of deficiencies in GCC optimizations with `libevas` on ARM, including GCSE, register allocation, autovectorization and loop prefetching, and proposed solutions to them, that altogether brought 15.78% average performance increase, and with up to 119% increase on certain tests, as measured with customized `expedite` benchmark. These results show that tuning existing GCC optimizations for specific platform and application may provide significant performance boost, comparable to that of developing a new compiler optimization.

1 Introduction

GCC is designed to be a multiplatform compiler. It also contains dozens of optimization passes that are parameterized in such a way that any target platform code can influence their decisions. Given the complexity of GCC, usually there is a room for improvement if you need to tune GCC for one particular target, while not paying attention to performance on other targets. The improvement may come from several sources. First, since most of the tuning happens for `x86` and `x86-64` architectures, there may be code generation deficiencies for a less popular target. A fix for them usually requires adding new instruction patterns or peephole optimizations to the backend, or adjusting target-dependent costs. For example, GCC inlining pass has internal constants specifying costs of function calls, prefetching pass has parameters that control cache metrics, etc. Second, machine-independent optimizations may not be tuned for a given target, meaning that their default behavior should be changed. For example, register allocator and/or inlining may use different limits on embedded targets than on general targets. Finally, a new target-independent feature could be implemented to take into account certain specific features of the target. For example, to make advantage of speculation feature of Intel Itanium, we have implemented its support in the instruction scheduler. This kind of improvements may take much time and resources.

In this work, our primary optimization target was ARM Cortex-A8 [1] architecture, including its NEON vector unit. In the paper we show a number of deficiencies that we found in GCC optimizations, and that are specific either to Cortex-A8 architecture or ARM platform in general. These optimizations include GCSE, prefetching, and autovectorization for NEON. We have proposed solutions to the problems found which altogether brought 15.78% average performance increase, with up to 119% gain on certain tests (as measured with customized `expedite` benchmark described in Section 2). We also show that using automatic compiler option tuning tools like ACOVEA may facilitate identification of those optimizations that need improvement as well as determining optimal optimization parameter values.

Rest of the paper is organized as follows. In Section 2 we give overview on `libevas` rasterization library, `expedite` benchmark test suite, ACOVEA tool and the environment we have used. In Section 3 we describe GCC optimizations we have analyzed and the improvements we made to them. In Section 4 we present the performance results of our optimizations and tuning. Section 6 outlines areas for the future work, and Section 6 concludes.

2 Test Application and Environment

As our primary performance target for GCC optimizations we have chosen `libevas`. It is a part of EFL (Enlightment Foundation Libraries) [2], which are base libraries for `E` window manager as well as other applications. This multiplatform library contains various routines for fast rasterization and processing image data, such as blending, scaling, clipping images, drawing polygons, etc. To measure `libevas` performance, we used a modified version of `expedite` benchmark suite, available in EFL repository. It was customized to improve results precision and speed, so it can provide results with variance less than 0.5% in 1-2 minutes test run time. The improvements to precision include adding few iterations to each test that are executed before time measurement is started so to exclude time required to fill cache from the result; a median filter is applied to evaluate final *fps* value for each benchmark among several runs. Better precision allowed us to significantly decrease the minimal number of iterations required for each test to obtain stable results. Also we have excluded from the original test set benchmarks with similar profiles and those with volatile results. To compute composite benchmark result value geometric mean is used. These improvements altogether allowed us to use this benchmark suite with automatic tuning tools as a complement to manual tuning.

To aid manual tuning, we used ACOVEA [3] automatic tuning tool. It aims to find a combination of options and parameters that provide best performance on a given benchmark using genetic algorithm [4, 5]. We have adapted ACOVEA to work in a cross environment. The changes include added capability to cross-compile benchmarks on `x86` machine, transfer binaries to `ARM` testboard, execute them, and transfer the resulting *fps* value back to `x86` host. Then, ACOVEA core cross-breeds GCC options from different runs according to performance

these options provide: the better *fps* the certain options combination achieves, the greater chance it has to reproduce.

Initially we ran ACOVEA tuning with a set of GCC flags that are enabled by default by `-O3` optimization level, plus `-fprefetch-loop-arrays` flag. It took about 4 days to complete with the following ACOVEA parameters: 20 generations, 3 populations, and 60 species in a population. This tuning run have shown that the greatest effect on performance has `-fno-gcse` option, which disables Global Common Subexpression (GCSE) optimization. This positive effect was observed on 44 out of 45 tests, and it didn't depend from other options specified along with `-fno-gcse`, or an input data. We give analysis of the GCSE optimization problem in Section 3.1. Other option combinations found by ACOVEA we have analyzed didn't tend to show consistent performance improvement, e.g. simply enabling `-fprefetch-loop-arrays` without tuning its parameters affected tests controversially, improving some by up to 10-15% while slowing down others as much as 25%.

Since there are more than 100 numeric parameters in GCC, each with its own integer value range, it is impractical to tune them all at once, so we selected few optimizations for detailed tuning of their parameters with ACOVEA. These optimizations are inlining, loop unrolling, register allocator and loop prefetching. We discuss results of this tuning in Sections 3.1–3.4, dedicated to corresponding optimizations.

In our work we used GCC 4.4.1 release branch as the base compiler.

3 GCC optimizations

In this section we discuss problems found in GCC optimizations and propose solutions for them.

3.1 GCSE

We have analyzed assembly code of `libevas` and identified a common deficiency in the way GCC deals with long immediate constants on ARM. On ARM, due to architecture constraints, a constant can be used as an immediate instruction operand if and only if it can be represented in the form `CONST_32 = CONST_8 << (2 * N)`, where `CONST_8` is an 8-bit constant, and $0 \leq N < 16$. If a constant doesn't comply with this constraint, the instruction can't use it as an immediate value, but should either preload it into a register or split original operation. Figure 1 shows an example of such constant splitting. Let's say we need the 2nd and 4th byte components from 4-byte integer variable *b* (this access pattern is very common in `libevas` blend routines). The C code to access appropriate components with a bit mask (at the left) is translated into two `bic` instructions (at the right). Assuming that in original application these instructions are located inside loop, in this case better solution would be store this constant into a register outside a loop and then use just one `and` instruction with that register

```
int a, b;
a = b & 0x00ff00ff;
```

(a) C code

```
ldr r3, [fp, #-8]       ; load b
bic r3, r3, #-16777216  ; r3 = r3 & ~0xff000000
bic r3, r3, #65280      ; r3 = r3 & ~0x0000ff00
str r3, [fp, #-12]      ; store a
```

(b) ARM assembly code generated by GCSE pass

Fig. 1. Splitting long ARM constants

as an operand instead of two `bic` instructions with immediate constants inside the loop.

We found that the main reason for generating such inefficient code is that during global common subexpression elimination (GCSE [6, 7]) optimization pass GCC doesn't consider ARM architecture specifics regarding immediate value representation in instruction code, assuming that constant propagation is always profitable, which is not true when propagating constants inside loops on ARM. There are three GCC passes that are involved in this problem: `pass_gcse`, `pass_rtl_move_loop_invariants`, `pass_split_all_insns` (by default executed in this order). At GCSE pass, two instructions `reg1 = const` and `reg2 = reg2 & reg1` are merged into `reg2 = reg2 & const`. At this stage, the compiler doesn't know whether `const` is a valid immediate constant for ARM or it needs splitting into two separate instructions and proceeds with merge anyway. Then, at `move_loop_invariants` pass, it wouldn't have any invariant to move, since at this point it is already an immediate value in instruction. Then, `split_all_insns` is run, which adds an extra instruction into the loop body. Changing the order of passes (doing loop invariant motion after split) doesn't help since after split it isn't a loop invariant any more because of data dependencies.

The problem can be worked around by disabling GCSE completely, but this isn't an appropriate solution, since in this case optimization opportunities can be missed. So we developed a solution for "conservative" GCSE, which takes into account ARM immediate value representation specifics. In order for loop invariant to work, we moved loop invariant code motion pass before GCSE pass, where all constants that could be moved outside of a loop body still reside in separate pseudo-registers. Our tests have shown that such pass order change doesn't affect the performance of `expedite` test suite. Loop invariant code motion pass has its own heuristics that estimate register pressure and doesn't allow moving invariant if it will likely result in a register spill. After loop invariants have been moved, our restricted GCSE will only allow to move "short" constants (those which don't require several operations to load) if moved into the loop body from the outside, and will allow GCSE to proceed in its usual way on the same loop hierarchy level. More strictly, at GCSE pass, we deny the transformation if the following two conditions are met:

1. The expression moved is a "long" 32-bit ARM constant, i.e. the constant doesn't fit into 12-bit immediate value (8-bit number and 4-bit shift values);

2. The expression is moved to destination block with a deeper loop hierarchy level than the source block, e.g. from an outer loop into a nested loop body.

Our patch includes two options to control GCSE behaviour on ARM. First option, `-farm-fix-gcse` checks just the first of above conditions, only denying transformation for long constants, and retaining original pass order; second, `-farm-fix-gcse-loop-hierarchy`, checks both conditions before allowing GCSE transformation and swaps GCSE and invariant code motion passes.

Restricting GCSE and letting loop invariant code motion pass to do its job increases performance of `libevas` on average by 5.5%. Though it isn't better than simple `-fno-gcse` for this application, we believe that it's the right way to address the problem and that there are applications that benefit from this approach. While testing our optimization on Aburto's benchmark suite, we found that it brings performance gain up to 10% on several tests without any significant regressions on others, compared to completely disabling GCSE which actually causes performance loss on this test suite. For example, on `hanoi` benchmark, completely disabling GCSE causes "short" constant 1 to be put into separate register, which results in additional register save and restore instructions in function prologue and epilogue. Since the subject function is recursive, these excessive instructions result in performance degradation by 10%, which is fixed by our patch that lets GCSE to propagate this "short" constant. We still need to test this optimization on more applications to make sure loop invariant code motion heuristics can handle well an increased number of loop invariants, so it doesn't cause performance regressions in loops with high register pressure.

3.2 The Register Allocator

Another problem we have found is excessive memory loads generated inside loops. Figure 2(a) shows the original code generated by GCC for a simple loop from `evas_common_scale_rgba_in_to_out_clip_smooth_c`. Both load instructions could be placed outside the loop, if there were enough free hardware registers available.

The reason for these excessive loads in the loop is that the cost of corresponding pseudo registers was calculated using basic block frequency via integer math, and truncating rounding caused sub-optimal code generation. We have tried a patch to use rounding to nearest integer in the register allocator, and it fixed the test case, but it did not provide performance improvements. We believe that due to the *NP*-complete nature of register allocation problem [6] this case represents just the bad case for register allocator heuristics, and in general it can be possibly improved by providing better estimation for basic block frequencies, which means using profiling information. Indeed, we have confirmed that when the compiler has precise execution counts from profiling, it generates exactly the same code for the problem loop either with or without the fix.

We have also tuned with ACOVEA the following GCC register allocator [9] options and parameters: `-fira-coalesce`, `-fira-algorithm`, `-fira-region`, `-fno-ira-share-spill-slots`, `-fno-ira-share-save-slots`, and `ira-max-`

```
.L133:
ldr lr, [fp, #-84]
mov r3, r1, asr #16
add r1, r1, r0
str r3, [lr, r2, asl #2]
ldr r3, [fp, #24]
add r2, r2, #1
cmp r3, r2
bgt .L133
```

(a) original code

```
.L133:

mov r3, r1, asr #16
str r3, [lr, r2, asl #2]
add r2, r2, #1

cmp r9, r2
add r1, r1, r0
bgt .L133
```

(b) code after a fix for register frequency rounding was applied

Fig. 2. Removing excessive invariant loads inside loops

`loops-num`. There were different option combinations found, that showed up to 6.5% gain on certain tests, while causing sometimes even bigger regression on others. Though only one option, `-fira-coalesce` seemed to improve performance consistently for the majority of the tests, giving 1-1.5% average gain. However, after we enabled `-fprefetch-loop-arrays` option later and tuned its parameters, the positive effect from `-fira-coalesce` was not longer reproduced. This shows that option tuning should involve all GCC options of interest at once, since optimizations tend to influence each other. At the same time, increased optimization search space may result in too much time to complete the tuning to make this approach practical.

3.3 Function Inlining

While tuning GCC inlining with ACOVEA, we found that `libevas` doesn't respond much to tuning its parameters, so we examined its source code to find whether there is a potential for this optimizations or it's just a problem with automatic tuning strategy that can't find the right parameters.

In `libevas` most CPU cycles are spent in tight loops performing rasterization. These loops are manually optimized by EFL developers, so this application has little inlinable calls that may affect the performance. We found that out of 19 EFL hottest functions that are invoked by `expedite` test suite 11 don't have calls at all, 4 have indirect calls through pointers, 1 is just a stub for `memcpy`, so function calls that can be inlined by GCC present only in 3 of these functions, which performance impact is minor. This way, the compiler options and parameters controlling inlining don't affect significantly the performance of `expedite` test suite, as we have found with automatic tuning, so `libevas` just might be not the right candidate to tune these optimizations. A good candidate for such study might be an application written in C++ that contains many small class member functions.

3.4 Loop Unrolling and Prefetching

As a part of original `libevas` hand-optimization, most critical loops are unrolled using custom `UNROLL8_PLD_WHILE` macro, which duplicates given loop body 8 times. This leaves compiler with little options for loop unrolling optimization: further unrolling such pre-unrolled loops usually doesn't yield any additional improvement. That's why automatic tuning of the RTL unroller parameters, similarly to inlining, didn't show significant improvement.

Modern ARM architectures have a *prefetching* feature, which allows preloading values from memory into L2 cache. This mechanism is controlled explicitly by a programmer or a compiler by issuing `pld` instruction, which hints CPU that data referenced by its argument soon will be needed, so CPU may start fetching it into its cache.

The abovementioned `UNROLL8_PLD_WHILE` macro, besides performing unrolling, inserts one `pld` prefetch instruction per unrolled loop body, assuming that cache line size equals to 32 bytes, and prefetching next cache line ahead of one iteration. Though this configuration shows +2.5% performance increase on `ARMv6`, it was found to be not optimal for `Cortex-A8`. Technical documentation for this architecture specifies L2-cache line size equal to 64 bytes, so each `pld` instruction generated with the macro hits the same cache line twice on `Cortex-A8`. Also, prefetching just 32 bytes ahead may be too little to allow complete loading next cache line before a new loop iteration begins. On the other hand, if the unrolled loop iterates just 4 times, long prefetching distance would be fetching values that will never be used.

We tried different prefetching parameters (distances in range from 32 to 320) and unrolling factors (from 2 to 16) and found that the best performance with this macro on `Cortex-A8` is achieved with no prefetching instruction at all and with unrolling factor equal to 4. These changes together yield increase of `libevas` performance by 6.5%, and the result can be evenly attributed to removing prefetching and changing unroll factor from 8 to 4. These results partially can be explained by the results of value profiling of `UNROLL8_PLD_WHILE` parameter `size`: about 20% of executed loops that are unrolled using this macro iterate just 4 times, and for about half of these loops the number of iterations doesn't exceed 16.

Prefetching is an optimization feature that is hard to implement properly, if it's done manually at the source code level, especially when it comes to tuning for different architectures at the same time. To benefit from this optimization, hardware cache specification should be taken into the account, such as L1 and L2 cache sizes, cache line size, the number of memory operations that can be processed simultaneously, and a latency of loading data from main memory into cache. GCC has a prefetching optimization (`-fprefetch-loop-arrays`) combined with loop unrolling, which takes these parameters into account in effort to generate optimal prefetching code.

GCC ARM backend doesn't override hardware cache parameters, so with this optimization common default GCC values are used, which were never tuned specifically for this architecture. However, as we found with `libevas` tuning,

these parameters should be set distinctly even among different ARM architectures. Not only these architectures have different latencies, cache sizes and limits on number of parallelly executed loads, but also `prefetch-latency` parameter, measured in number of instructions executed before prefetch operation is completed, has different meaning depending on instruction latencies and an issue rate (e.g. `Cortex-A8` is a dual-issue, while `ARMv6` is single-issue architecture).

We tried to specify cache parameters found in ARM technical specification (`l1-cache-line-size=64`, `l1-cache-size=32`, `l2-cache-line-size=256`) as well as tuning them with ACOVEA. Specifying correct parameters from documentation does improve the performance, but with automatic tuning we have found other parameter sets that slightly differ from those in technical specs but give even greater improvement. Here are two best parameter strings found by ACOVEA, each of them is beneficial for distinct subset of benchmarks:

1. `l2-cache-size=256 l1-cache-size=16 simultaneous-prefetches=8 prefetch-latency=200 l1-cache-line-size=32`
2. `l2-cache-size=512 l1-cache-size=64 simultaneous-prefetches=6 prefetch-latency=400 l1-cache-line-size=64`

The second parameter set provides slightly better overall performance, so in final results table we use the latter. These two parameters have one common property: the `simultaneous-prefetches` parameter is set far above GCC default value of 3.

Some parameters found by ACOVEA differ from those in hardware specification, e.g. the best value found for `l1-cache-line-size` is 32, though ARM documentation specifies line size equal to 64. Smaller cache line value causes prefetch optimization to choose smaller unrolling factor (since it tries to issue one prefetching instruction per unrolled loop body). So the fact that ACOVEA has found cache line size to be less than that in specification shows that for some tests smaller unrolling factor is better than not have an extra preload instruction, which hits the same cache line twice.

Also, it can be noted that prefetching optimization with parameters properly adjusted is overall 4% better than without prefetching and with just 4 regressions in range 2.73%, opposed to default prefetching parameters which yield 9 regressions in that range and two 13% regressions, while gaining just 2% on average. Prefetching with parameters properly tuned speeds up certain tests by as much as 20%.

3.5 Autovectorization for NEON

GCC features autovectorization for many SIMD architectures [8], including NEON vfpu that is available in Cortex-A8. We studied how well this feature works with `libevas` code. After enabling autovectorizer (`-ftree-vectorize -mfpu=neon -mfloat-abi=softfp`), we were surprized to observe performance regression. Since the target application spends most of its runtime in tight rasterization loops, supposedly it should have respond well to vectorization.

```
int main() {
  int a[256], b[256];
  int i;

  for (i = 0; i<256; i++) {
    a[i] = b[i] >> 8;
  }
}
```

(a) original code

```
.L2:
  add     r2, r0, r3
  fldd    d16, [r2, #0]
  vmov.32 r2, d16[0]
  vmov.32 r1, d16[1]
  mov     r2, r2, asr #8
  str     r2, [r5, r3]
  add     r2, r5, r3
  add     r3, r3, #8
  mov     r1, r1, asr #8
  cmp     r3, #1024
  str     r1, [r2, #4]
  bne     .L2
```

(b) Original assembly code generated by GCC

```
.L2:
  add     r2, r5, r3
  add     r1, r0, r3
  add     r3, r3, #8
  cmp     r3, #1024
  fldd    d16, [r1, #0]
  vshl.s32  d16, d16, d17
  fstd    d16, [r2, #0]
  bne     .L2
```

(c) Assembly code after NEON backend was fixed

Fig. 3. Autovectorization of shift operation on NEON

First, we have analyzed why so few loops (just about 25%) were vectorized automatically by GCC. Most common causes of autovectorizer's failure were the following: function calls within the loop body (mostly indirect calls, so they can not be inlined), switch operator within a loop, and unsupported operations (e.g. there is no support for vector division on NEON). It's worth to note that switch operators within loops in `libevas` are used to specialize two cases for transparency values 0 and 1, so a multiplication by alpha-channel value could be replaced with simple copy of either source or destination color value. Though such specialization prevents the loop from being vectorized for NEON, pure ARM specialized code still significantly outperforms autovectorized NEON code on `expedite` tests.

We have found a problem with autovectorization of shift operations for NEON. If a loop being autovectorized contains shift operations (`>>`), autovectorizer is not able to find appropriate vector shift operation, so loop is vectorized partially: for all the rest operations (except shifts) vector instructions are generated, but for shifts data is moved first from NEON vector registers to ordinary ARM 32-bit ones, then ARM shifts for each vector component are issued, and finally data is moved back to NEON registers to store the vector into memory. Such transfers from NEON to ARM core and back cause severe performance degradation of the affected loops. Figure 3 gives an example of such poor loop auto-vectorization.

The cause for this problem was a bug in ARM NEON backend, which assigned shift operations to wrong operation table. Fixing this issue improved overall `expedite` performance by 8.82%, while certain tests ("Rect Blend" family), which suffered the most from poor shifts vectorization, grow as much as by 171%. There are few regressions, but most of them are within an error margin.

Also, we have found that specifying `-mvectorize-with-neon-quad` option gives slightly better overall results (about 1%) than default double-integer vectorization.

4 Experimental results

The performance results on reduced `expedite` test suite (as described in Section 2) are presented in Table 1. These results were obtained on `EBV Beagle` board with `vga` profile and using linux framebuffer. All the values presented are medians among 3 runs of the whole test suite. Due to space constraints, we omit those tests which performed similarly across all optimizations.

We reference each column with corresponding number. In the first row we specify the options used for benchmarks, or reference with square brackets another column where these options can be found, and specify just those options in which they differ from the referenced column, e.g. *"[A2] no prefetch, unroll=4"* means *"the same compile parameters as for A2, but with manual prefetching in macro turned off and unrolling factor set to 4"*.

The first column of Table 1 (`A1`) gives numbers for base GCC optimization level, `-O2`. We have chosen `-O2` as the base, since our tests have shown that `-O2` outperforms `-O3` on `expedite` by 0.5-1%. The second column, `A2`, shows results for base optimization level with GCSE turned off. It can be seen that this option is good for almost every test, since in `libevas` long constants, whose performance is highly affected by GCSE, are widely used in color component masks (like `0xFF00FF00`). The next column, `A3` shows our efforts on fixing GCSE to work properly on ARM rather than disabling it completely. Though currently with `libevas` it doesnt show additional gain relative to `-fno-gcse`, we still believe that for other applications this approach may be more profitable than disabling GCSE, as we have seen 10% gain on Aburto's `hanoi` benchmark. Still, this optimization delivers performance 5.5% better than `-O2`. The next column, `A5`, shows results for adjusting unrolling factor from 8 to 4 in `UNROLL8_PLD_WHILE` macro and disabling there manual prefetching (each of them contributed approximately by 3%). The next two columns, `A6` and `A7`, show numbers for prefetching/unrolling optimizations (`-fprefetch-loop-arrays`). This GCC optimization with default parameters gives 2% average improvement, but with few serious regressions up to -13%. If parameters are tuned properly (we used the second parameter string from Section 3.4), this optimization provides 4% improvement (`A7`), while growth is more evenly distributed among tests and without significant regressions. Total average gain on customized `expedite` from all ARM optimizations developed/tuned comparing to base `-O2` level is 15.78%.

Due to space constraints, we don't show separate results table for NEON autovectorization. The main results for NEON autovectorization are as follows. Fixing the autovectorization of shifts improved overall `expedite` performance by 8.82%, while certain tests ("Rect Blend" family), which suffered the most from poor shifts vectorization, grew as much as by 171%. These results are achieved with `-mvectorize-with-neon-quad` option, which gives about 1% overall gain, and manual unroll factor set to 4 in macro. The manual unrolling factor setting makes sense for those loops that don't get autovectorized (e.g. due to the presence of switch operator). Still, performance on pure ARM (without NEON) is 1.5% better than that with NEON autovectorization due to unaligned data accesses that autovectorizer currently doesn't handle.

	-O2 (A1, base)	-O2 -fno-gcse (A2)	to base, %	-O2 -farm-fix-gcse (A3)	to base, %	[A2] no prefetch, unroll =4 (in macro) (A5)	to [A2] -fno-gcse, %	[A5] -fprefetch-loop-arrays (A6)	to [A5], %	[A6] + prefetching params (A7)	to [A6] (default prefetch params), %	to [A5], %	to [base] -O2, %
1 - Widgets File Icons	11.53	11.79	**2.25**	11.9	**3.21**	12.35	**3.87**	13.26	**7.37**	13.07	**-1.43**	**5.83**	**13.36**
2 - Widgets File Icons 2	23.81	24.07	**1.09**	23.98	**0.71**	26.85	**12.02**	31.19	**16.16**	30.05	**-3.66**	**11.92**	**26.21**
5 - Image Blend Unscaled	14.89	14.9	**0.07**	15.47	**3.90**	16.46	**10.40**	16.63	**1.03**	16.89	**1.56**	**2.61**	**13.43**
6 - Image Blend Solid Middle Unscaled	10.92	11.06	**1.28**	11.12	**1.83**	11.65	**4.48**	11.78	**1.12**	11.75	**-0.25**	**0.86**	**7.60**
7 - Image Blend Fade Unscaled	7.3	7.92	**8.49**	7.81	**6.99**	8.3	**5.60**	8.15	**-1.81**	8.35	**2.45**	**0.60**	**14.38**
9 - Image Blend Solid Unscaled	50.71	52.01	**2.56**	51.58	**1.72**	50.37	**-3.25**	50.91	**1.07**	52.54	**3.20**	**4.31**	**3.61**
10 - Image Blend Solid Fade Unscaled	10.44	11.73	**12.36**	11.63	**11.40**	12.53	**6.10**	12.38	**-1.20**	12.54	**1.29**	**0.08**	**20.11**
11 - Image Blend Solid Fade Power 2 Unscaled	10.46	11.74	**12.24**	11.63	**11.19**	12.55	**6.18**	12.38	**-1.35**	12.54	**1.29**	**-0.08**	**19.89**
12 - Image Blend Nearest Scaled	6.41	6.48	**1.09**	6.73	**4.99**	7.09	**9.92**	7.4	**4.37**	7.45	**0.68**	**5.08**	**16.22**
14 - Image Blend Smooth Scaled	1.37	1.45	**5.84**	1.45	**5.84**	1.47	**7.30**	1.43	**-2.72**	1.43	**0.00**	**-2.72**	**4.38**
16 - Image Blend Nearest Same Scaled	22.81	23.91	**4.82**	23.86	**4.60**	24.62	**1.53**	25.12	**2.03**	25.05	**-0.28**	**1.75**	**9.82**
17 - Image Blend Nearest Solid Same Scaled	63.29	64.85	**2.46**	64.25	**1.52**	63.84	**-1.51**	65.43	**2.49**	66.1	**1.02**	**3.54**	**4.44**
18 - Image Blend Smooth Same Scaled	22.87	24.22	**5.90**	23.71	**3.67**	24.74	**2.02**	25.13	**1.58**	25.1	**-0.12**	**1.46**	**9.75**
19 - Image Blend Smooth Solid Same Scaled	69.94	71.61	**2.39**	71.31	**1.96**	69.36	**-4.01**	70.91	**2.23**	73.27	**3.33**	**5.64**	**4.76**
20 - Image Blend Border	1.47	1.56	**6.12**	1.56	**6.12**	1.62	**10.20**	1.58	**-2.47**	1.58	**0.00**	**-2.47**	**7.48**
21 - Image Blend Solid Middle Border	14.15	14.61	**3.25**	14.61	**3.25**	14.82	**4.15**	14.83	**0.07**	14.84	**0.07**	**0.13**	**4.88**
22 - Image Blend Solid Border	20.69	21.57	**4.25**	21.59	**4.35**	21.73	**3.97**	21.52	**-0.97**	21.74	**1.02**	**0.05**	**5.07**
23 - Image Blend Border Recolor	1.4	1.49	**6.43**	1.48	**5.71**	1.51	**9.42**	1.49	**-1.32**	1.49	**0.00**	**-1.32**	**6.43**
25 - Image Data ARGB	48.18	47.74	**-0.91**	47.07	**-2.30**	47.64	**1.21**	53.73	**12.78**	55.33	**2.98**	**16.14**	**14.84**
26 - Image Data ARGB Alpha	18.18	18.41	**1.27**	18.31	**0.72**	19.94	**8.90**	24.02	**20.46**	22.2	**-7.58**	**11.33**	**22.11**
27 - Image Data YCbCr 601 Pointer List	31.04	31.88	**2.71**	31.18	**0.45**	31.89	**2.08**	32.55	**2.07**	33.16	**1.87**	**3.98**	**6.83**
28 - Image Data YCbCr 601 Pointer List Wide Stride	25.95	27.51	**6.01**	27.13	**4.55**	27.57	**6.12**	27.13	**-1.60**	28.29	**4.28**	**2.61**	**9.02**
29 - Image Crossfade	33.21	34.6	**4.19**	34.36	**3.46**	34.85	**1.46**	45.6	**30.85**	41.99	**-7.92**	**20.49**	**26.44**
30 - Text Basic	38.4	38.89	**1.28**	38.74	**0.89**	39.74	**4.44**	40.68	**2.37**	41.31	**1.55**	**3.95**	**7.58**
31 - Text Styles	3.76	3.8	**1.06**	3.82	**1.60**	3.93	**3.69**	3.94	**0.25**	3.97	**0.76**	**1.02**	**5.59**
33 - Text Change	19.8	19.89	**0.45**	19.56	**-1.21**	20.93	**6.24**	21.58	**3.11**	21.79	**0.97**	**4.11**	**10.05**
34 - Rect Blend	5.76	8.24	**43.06**	8.19	**42.19**	10.73	**30.22**	9.32	**-13.14**	12.61	**35.30**	**17.52**	**118.9**
36 - Rect Solid	44.63	46.73	**4.71**	45.9	**2.85**	50.47	**9.55**	52.88	**4.78**	53.13	**0.47**	**5.27**	**19.05**
37 - Rect Blend Few	711.5	824.2	**15.84**	818.5	**15.04**	923.4	**14.51**	889.4	**-3.69**	943.8	**6.12**	**2.21**	**32.65**
39 - Rect Solid Few	1066	1096	**2.80**	1088	**2.05**	1204	**12.33**	1230	**2.14**	1176	**-4.37**	**-2.33**	**10.30**
41 - Image Blend Occlude 2 Few	131.1	133.8	**2.09**	136.1	**3.81**	139.6	**4.97**	143.2	**2.59**	146.8	**2.55**	**5.20**	**12.01**
43 - Image Blend Occlude 1 Many	64.13	67.64	**5.47**	67.86	**5.82**	68.92	**3.51**	68.31	**-0.89**	69.28	**1.42**	**0.52**	**8.03**
45 - Polygon Blend	11.1	13.44	**21.08**	12.67	**14.14**	13.65	**4.04**	13.33	**-2.34**	15.16	**13.73**	**11.06**	**36.58**
Geometric Mean	22.97	24.32	**5.88**	24.23	**5.48**	25.55	**6.47**	26.06	**1.97**	26.60	**2.09**	**4.09**	**15.78**

Table 1. The performance results (frames per second)

We have also verified the results against full original `expedite` test suite (as found in EFL repository) with 11.12% average performance optimization and with two different Cortex-A8 boards. Also, we verified the results with a different input data set, getting about 8% gain on original `expedite`.

5 Future work

Though we have fixed the vector shift instructions, there are still problems remaining with this optimization. First, autovectorizer currently is unable to produce operations involving vector and scalar arguments at the same time, e.g. for original operation `a[i] << CONST` it would first produce a vector containing four identical constants, and only then issue a vector operation `a_vec[j] << 4xCONST`, though NEON has a distinct operation for shifting vector by a single scalar. Second, the vectorizer can only handle aligned data, and if at runtime it finds out it's misaligned, it executes regular non-vectorized version of a loop, thus wasting time on alignment checks. We believe that this GCC optimization has more potential, and additional efforts should be done in improving autovectorization for NEON.

The results of tuning GCC loop prefetching/unrolling optimization prove that it is important for achieving good performance on ARM in applications with intensive memory usage and regular memory access pattern (like rasterization routines in `libevas`). Its performance results may vary among applications (as they vary among `expedite` tests), from input data (with small data sizes prefetching won't have time even to complete a load of first portion of data since prefetch distance may be greater than data size), and from the target architecture. We think that this optimization needs more detailed manual analysis so to find whether it has some implementation specifics that can be improved on ARM, as well as automatic tuning of its parameters with more applications.

Also, more general mechanism should be developed to provide GCSE optimization with a target-specific information on representation of constants (e.g. via a target hook), so to improve code on other architectures that may have similar constraints to those found on ARM.

6 Conclusions

We have identified a number of performance regressions in GCC optimizations with `libevas` on ARM, including GCSE, register allocation, autovectorization and loop prefetching, and suggested fixes to them. The solutions proposed altogether brought 15.78% average performance increase as measured with reduced `expedite` benchmark (as described in Section 2), with up to 119% increase on certain tests. We have verified the produced speedup on the full benchmark with 11.12% average performance optimization and with up to 119% performance increase on some tests.

Overall, we believe that a project on optimizing GCC compiler for certain applications and target architecture (both manually and automatically) makes

perfect sense, as such projects identify weak places of the compiler with regard to this target and application, and fixing these places may bring performance improvements. The results of such optimization should be made available to GCC community and developers of the target application so to avoid duplicate work on the same compiler optimizations and to encourage further development of the application using coding practices found to help compiler optimizations. After optimizations developed will be verified on wider range of applications, they could be enabled by default in GCC compiler for this target platform.

References

1. ARM Cortex-A8 Technical Reference Manual. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344d/DDI0344D_cortex_a8_r2p1_trm.pdf`
2. The Enlightenment Foundation Libraries (EFL) web page. `http://www.enlightenment.org/p.php?p=about/efl`
3. ACOVEA home page. `http://www.coyotegulch.com/products/acovea/index.html`
4. Christopher G. Langton, ed. Artificial Life: An Overview. MIT, 1997.
5. David E. Goldberg. Genetic Algorithms in Search, Optimization & Machine Learning. Addison-Wesley, 1989.
6. A. Aho, R. Sethi, J. Ullman. Compilers: Principles, Techniques and Tools. Addison-Wesley, 1988.
7. E. Morel, C. Renvoise. Global Optimization by Suppression of Partial Redundancies. In Communications of the ACM, Vol. 22, Num. 2, Feb. 1979.
8. D. Nuzman, R. Henderson. Multi-platform Auto-vectorization. In Proceedings of the International Symposium on Code Generation and Optimization, 2006.
9. V. Makarov. The integrated register allocator for GCC. In Proceedings of the GCC Developers Summit, July 2007.

Portable and Efficient Auto-vectorized Bytecode: a Look at the Interaction between Static and JIT Compilers

Erven Rohou – HiPEAC member

INRIA, Rennes, France

Abstract. Heterogeneity is a confirmed trend of computing systems. Bytecode formats and just-in-time compilers have been proposed to deal with the diversity of the platforms. By hiding architectural details and giving software developers a unified view of the machine, they help improve portability and manage the complexity of large software.
Independently, careful exploitation of SIMD instructions has become crucial for the performance of many applications. However, auto-vectorizing compilers need detailed information about the architectural features of the processor to generate efficient code.
We propose to reconcile the use of architecture neutral bytecode formats with the need to generate highly efficient vectorized native code. We make three contributions. 1) We show that vectorized bytecode is a viable approach that can deliver portable performance in the presence of SIMD extensions, while incurring only minor penalty when SIMD is not supported. In other words, the information that a loop can be vectorized is the vectorized loop itself. 2) We analyze the interaction between the static and just-in-time compilers and we derive conditions to deliver performance. 3) We add vectorization capabilities to the CLI port of the GCC compiler.

1 Motivations

In this study, we attempt to reconcile two apparently contradictory trends of computing systems. On the one hand, hardware heterogeneity favors the adoption of bytecode format and late, just-in-time (JIT) code generation. On the other hand, exploitation of hardware features, in particular SIMD extensions, is key to extract performance from the hardware.

1.1 Mobile Long-lived Applications and Processor Heterogeneity

Heterogeneity of computing systems is a global trend. On embedded systems, this trend has been driven by the drastic constraints on cost, power and performance. General purpose computers also feature some degree of variability: availability of a floating point unit, width of the vectors of the SIMD unit, number of cores, kind and features of the GPU, and so on. Some predict that technology variability will make it hard to produce homogeneous manycores, and that the large number of available cores will push to specialize cores for dedicated tasks [7].

The lifetime of applications is much longer than that of the hardware. This problem is known as *legacy code* in the industry. Embedded systems rarely offer binary compatibility because of the associated cost. Servers and personal computers usually do, but at a significant design cost (with occasional disruptions like DEC, or Apple). However, compatibility is limited to functionality; old code can only take advantage of increased clock frequency (which, incidentally, has recently stopped) and improved microarchitecture, but not of additional features or increased parallelism. Applications also become mobile. Because of the ubiquity of computing devices, application developers must make sure that their applications run on dozens of platforms, some being unknown or not completely specified.

Bytecode formats and just-in-time compilers have been proposed to deal with heterogeneity. Bytecode can be deployed to any system as long as a JIT compiler is available for each core on which the code is going to run. Application developers do not even have to know the hardware on which their code will eventually run. Processor virtualization, i.e. virtual machine and JIT compilers, is a mature and widely spread technology: Java applications can be found from games in cell phones to web servers and banking applications, and CLI [8] (the core of the .NET initiative [16]) is growing fast. Virtualization can address the above mentioned problems: it reduces the burden put on software developers who no longer need to deal with varying hardware, it guarantees that application lifetimes can span several generations of hardware, and, to some degree, it makes it possible for old code to exploit new hardware features.

1.2 Exploitation of Word-Level Parallelism

The careful exploitation of SIMD instructions is crucial for the performance of many applications. All major instruction sets provide SIMD extensions (SSE on x86 processors, Altivec on PowerPC, VIS on Sparc, etc.), and keep adding new vector instructions (SSE4.1, SSE4.2, SSE4a). Even though significant progress has been made in the recent years, good auto-vectorization is still an open difficult problem. This is illustrated by the abundance of literature [1, 3, 14, 17], the ongoing work in open source compilers like GCC, or simply the existence of source-level *builtins* that let programmers insert instructions by hand when the compiler fails to detect a pattern.

Auto-vectorization is a complex optimization for several reasons:

- strong conditions must be met by the code, in particular in terms of data dependencies;
- for better applicability, one wants the optimization to also apply to outer loops and to handle strided accesses and other complex patterns;
- each particular instruction set has a very specific set of vector instructions, and associated constraints [17]: required alignment, available registers, etc.

1.3 Putting Things Together

If static compilers have a hard time vectorizing loops, the situation is much worse for JIT compilers. A simple look at the vectorizer in GCC gives an idea — more than 20,000 lines, not counting data dependence analysis and the construction of the SSA form. The complexity of the analysis and transformation makes the vectorizer

unfit for JIT compilers which are often running on memory and CPU constrained environments.

Conversely, *statically* auto-vectorizing loops for a bytecode representation is challenging because the actual features and constraints of the execution platform are unknown at compile-time. At run-time, SIMD extensions might not be available.

In this paper, we investigate how processor virtualization and auto-vectorization can be reconciled. We make three contributions:

1. we show that vectorized bytecode is a viable approach, that can yield the expected speedups in the presence of SIMD instructions, and a minor penalty in its absence;
2. we measure the performance of loop kernels on several architectures, we analyze the interaction between the static and the JIT compilers and we provide suggestions for a good performance of vectorized bytecode;
3. we describe our modifications to the CLI port of the GCC compiler to emit vectorized bytecode, and we make them publicly available in the GCC repository.

This paper is organized as follows. Section 2 presents the high level view and rationale of our approach, while Section 3 goes over the details of the implementation. We develop our experiments and our analyses in Section 4. Related work is reviewed in Section 5 and we conclude in Section 6.

2 Reconciling Processor Virtualization and Auto-vectorization

This paper uses or refers to compilation and optimization techniques, such as function inlining, loop unrolling, data dependence analysis, SSA form. We did not introduce any new technique *per se*, but rather we take them for granted and we use them. The interested reader can refer, for example, to [15].

2.1 Split-Compilation

Our proposal builds on top of the idea of split-compilation. Split-compilation refers to the fact that a given source code undergoes two compilation steps before it becomes machine code.

1. The first step translates source code to bytecode. This happens on the programmer's workstation. This means that the resources available to the compiler are virtually unlimited: gigahertz, gigabytes and minutes of compile-time are common. However, no assumption can be made on the actual platform on which the application will eventually run.
2. The second step converts the bytecode to machine code. It happens *just-in-time*, i.e. on the final device and at run-time. Resources are likely to be limited, especially on an embedded system like a cell phone or a DVD player. Compile-time is also visible to the end-user, and thus it must be kept as small as possible.

The key of split compilation is to move as much complexity as possible from the second step to the first one [19]. The first pass is in charge of all target independent optimizations. Target specific optimizations obviously cannot be applied. Expensive analyses, however, can be run, and their results encoded in the bytecode, so that the JIT compiler can directly benefit from their outcome.

2.2 Vectorized Bytecode

Previous work by Leśnicki et al. annotated the bytecode to mark the variables and types of interest to the JIT compiler (see Section 5 for more details). We believe that this kind of annotation was rather difficult to generate and left too much work to the online vectorizer. Instead, we choose a more drastic approach: the information that a loop can be vectorized is the vectorized loop itself. All the expensive loop transformation is done in the first pass, and we make sure that it can be undone at a low cost if necessary. It is cheaper (at run-time) to undo a speculative vectorization than to do it when necessary.

As further explained in Section 3, we base our work on the CLI format. However, it is very important that we do not extend the format itself. The vectorized bytecode we produce must run unmodified on any CLI compliant virtual machine. Vector information is expressed by means of new types and methods. Vector operations in the user code appear as method invocations.

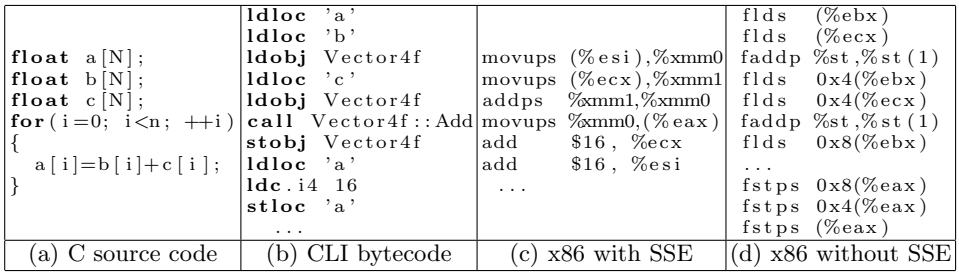
We need to achieve three objectives:

1. in the presence of SIMD extensions in the instruction set, a JIT compiler aware of our optimization must produce fast machine code;
2. in the absence of SIMD extensions, or when using any other JIT compiler, the code must run correctly; and
3. the penalty when running vectorized bytecode without SIMD support (whether in the JIT compiler or in the instruction set) must be minimum.

All three objectives are made possible by the dynamic code generation mechanism. Point 1 is "simply" a different code generation. The static and JIT compilers must agree upon a naming convention for special types and methods. When a call to such a method is encountered, a specialized instruction pattern is emitted, instead of a call. Figure 1 illustrates this for a simple vector addition. Column (a) shows the C code for this simple loop. Column (b) shows a part of the loop once translated to CLI bytecode. Bear in mind that the execution model relies on an execution stack. `Vector4f` is a type defined in a library that represents a vector of four single precision floating point numbers. `ldloc 'b'` places the address of a vector element on the stack. `ldobj` consumes the address and loads the element on the stack. The same is true for element `c`. The method `Add` is then called to perform the addition. The result is stored at the address initially pushed on the stack by the `ldloc 'a'` instruction. The remaining instructions increment the induction variable `a`. When translating this bytecode (column (c), showing x86 assembly code [10]), the JIT compiler recognizes the type `Vector4f` and it emits a `movups` instruction that targets an SSE register. Similarly, `Add` is recognized, and a single `addps` instruction is emitted.

Point 2 consists in providing a library which implements all the functions defined by the naming convention. A compiler unaware of the special semantics will emit regular code, the same way as any other code.

Point 3 combines the just-in-time code generation with inlining. We make the assumption that JIT compilers always have the capability to inline functions. This is not a strong assumption because bytecode and JIT compilers were initially designed for object oriented languages, which tend to encourage small functions, including accessors (setters/getters) and constructors. Inlining has been key for performance



```
float a[N];
float b[N];
float c[N];
for(i=0; i<n; ++i)
{
  a[i]=b[i]+c[i];
}
```

```
ldloc 'a'
ldloc 'b'
ldobj Vector4f
ldloc 'c'
ldobj Vector4f
call Vector4f::Add
stobj Vector4f
ldloc 'a'
ldc.i4 16
stloc 'a'
  ...
```

```
movups (%esi),%xmm0
movups (%ecx),%xmm1
addps  %xmm1,%xmm0
movups %xmm0,(%eax)
add    $16, %ecx
add    $16, %esi
 ...
```

```
flds  (%ebx)
flds  (%ecx)
faddp %st,%st(1)
flds  0x4(%ebx)
flds  0x4(%ecx)
faddp %st,%st(1)
flds  0x8(%ebx)
...
fstps 0x8(%eax)
fstps 0x4(%eax)
fstps (%eax)
```

(a) C source code | (b) CLI bytecode | (c) x86 with SSE | (d) x86 without SSE

Fig. 1. Code generation schemes

since the beginning of this technology. The vector operations provided in the library are very basic, they include arithmetic, constructors, and load or store operations. They are good candidates for inlining. The end result after minimal cleanup is code similar to the column (d) of Figure 1. The vector operations are effectively unrolled by an amount equal to the width of the vector. Unrolling is known to help performance at the expense of code size. However, we do not expect any significant code bloat because only the small loops corresponding to vector operations are unrolled.

Vectorizing the bytecode gives another advantage: in cases when the static compiler is not able to generate the SIMD instructions, programmers can still manually insert builtins in the source code, as they usually do for performance critical loop nests. It makes no difference to the JIT compiler whether those builtins were automatically generated or hand written.

2.3 Other Design Decisions

SIMD instruction sets vary a lot in number of supported idioms, expressiveness, and constraints. Many choices can be made to best match the abstract vector representation of the bytecode to all possible instances of vector instruction sets. Because we rely on an existing compiler (see Section 3), our choices are limited and we mostly follow the decisions made in the GCC GIMPLE representation. For further details about those design decisions, we refer to the discussion "Generality vs. applicability" of [17].

Alignment Alignment constraints and realignment idioms are a typical burden of vectorizing compilers. We face the additional problem that the static compiler does not know whether the target supports unaligned accesses. We have two options.

- We support unaligned accesses in the bytecode. The static compiler generates simpler code. It is up to the JIT compiler to realign memory accesses if needed.
- Or we require aligned memory accesses in the bytecode. In this case, the static compiler generates the realignment code in the bytecode. The JIT compiler is guaranteed to see only aligned memory accesses.

We decided for the former approach, because it generates simpler code. The latter, while always correct, requires extra work from the JIT compiler when misaligned accesses are available: it needs to eliminate redundant checks or even entire loops that

were generated to realign accesses by peeling some iterations off the main computation loop. In the former approach, the static compiler can pass alignment information to the JIT compiler, so that no unnecessary realignment is generated when arrays are known to be properly aligned.

Vector Width Vector width is another parameter dictated by the architecture, hence unknown in the static compiler. We take the following approach: since most architectures have 128-bit wide vector operations, this is the width we vectorize for. An architecture with a different vector width, like the upcoming AVX or Larrabee) will fall back on the scalar implementation as described in this section. A smarter JIT compiler could try adjust to the actual width, but this needs additional data dependence analysis at run-time, or extra annotations that specify the maximum vectorization factor for each loop.

Multiversioning We could have made the choice to generate two versions of each loop: a vectorized one, and a scalar one. The consequence, however, is that the static compiler should use a reduced set of vector instructions, unless it runs the risk that the vectorized code is too specialized and never runs on many architectures. Another option is to generate more than two versions of the same loop (a technique generally called multiversioning), to adjust to most targets. Obviously the cost is code size increase. Generating several versions of each vectorizable loop in not even an option for embedded systems, for example.

Our approach has the advantage that it exposes all the opportunities to the JIT compiler in a single version of the loop, while letting it gracefully handle the patterns that do not have hardware support.

3 Implementation

Proper evaluation of our proposal requires aggressive static and JIT compilers to make sure that results are not biased because of poor optimizations unrelated to our focus. Compilers — static and JIT — are huge pieces of software. We leveraged two open-source software projects: GCC for the static compiler, and Mono for the virtual machine and JIT compiler.

We chose to implement our experiments in the GCC compiler for several reasons beyond the availability of the source code. The good quality of the generated code makes the results trustworthy, and the well documented auto-vectorizer [18], despite its internal complexity, is easy to retarget thanks to the GCC machine model.

We have shown that the CLI format is appropriate for deployment onto embedded systems [4, 6]. CLI is a standard format [8], which means that it is more likely to be portable to various architectures. In fact, several commercial and open-source projects already provide execution environments for the CLI format (see Section 5).

We previously developed a GCC back-end for the CLI format [5, 20], however with very limited support for vector types. One of the contributions of this work is to add vectorization capabilities to the CLI back-end. It is publicly available in the branch `st/cli-be` of the GCC repository.

3.1 Machine Model Technicalities

Activating the GCC vectorizer consists in modifying a few places in the machine description files. First we need to globally instruct the compiler that vector modes are supported by defining the function `cil32_vector_mode_supported_p`.

Then, we need to provide the width of the available vector types. This is accomplished through the macro `UNITS_PER_SIMD_WORD`.

Ideally, the CLI machine description does not need any register at all, since operations are carried on the evaluation stack, and the set of local variables (CLI *locals*) used to store values is infinite, making register allocation a non-issue. Still, GCC needs a minimal set of registers for its own mechanics. In particular, the largest *mode* that can be produced by the vectorizer is computed from the largest set of contiguous registers in the same class. The existing machine description defined only one 32-bit register, thus making vectorization impossible. We increased the number of available registers by modifying the macros `FIXED_REGISTERS` and `CALL_USED_REGISTERS`. We also modified `FIRST_PSEUDO_REGISTER` accordingly.

Finally, we simply add the definition of the supported vector modes and all the supported arithmetic instructions to the machine description file `cil32.md`, as well as the special `movmisalign` instruction used by GCC to generate misaligned accesses.

3.2 Intermediate Representation

We keep the GIMPLE representation, and the vectorizer as a whole, unmodified. Vector types are produced by the vectorizer as usual, based on the information derived from the machine model. The differences appear in the stack based intermediate representation (IR), introduced in [20] to replace RTL in the CLI back-end. This representation was shown to be better for emitting CLI for two main reasons: it has a concept of evaluation stack, and it is strongly typed, a necessary condition to emit correct CLI.

Most vector operations are eventually translated in the bytecode as calls to well defined library functions (builtins). However, function calls tend to make the code more difficult to analyze and optimize. For this reason, we try to postpone the emission of the builtins as much as possible. In particular, we rely on the existing policy in GIMPLE and the stack-based IR assuming that arithmetic operators are polymorphic: we do not add any new arithmetic nodes operating on vectors. Rather, we extend the semantics of existing operators to accept the new vector types.

We handle the vector constructors (the `CONSTRUCTOR` GIMPLE node), with a new IR statement named `VEC_CTOR`. To distinguish vector loads and stores, we also introduce a `LDVEC` and a `STVEC` instruction. Similarly to GIMPLE, we introduce new statements for operators which do not have any scalar equivalent, for example the dot product, or the saturating arithmetic.

We add a pass just before the CLI emission to recognize the vector statements and to transform them to calls to builtins. This pass walks over all the statements of a function, and computes the status of the stack before each of them (this is always possible without dataflow analysis thanks a special constraint of the CLI format, see § III, 1.7.5 of [8]). Arithmetic statements that operate on vector types are replaced by the corresponding builtin. Constructors and `LDVEC STVEC` are rewritten.

3.3 Execution Environment

We used Mono [13] as our execution platform. Mono is an open development initiative to develop a UNIX version of the Microsoft .NET environment. It contains a CLI virtual machine — complete with a class loader, a JIT compiler and a garbage collector — as well as a class library and a compiler for the C# language. In this project, we rely on the JIT compiler of the VM.

The Mono environment contains the library Mono.Simd.dll. It defines all the 128-bit vectors types (four single precision floats, four 32-bit integers, eight 16-bit integers, etc.), and the basic arithmetic operations on them. A source implementation of the types and methods is provided in C#, and compiled to CLI. This code is used as a backup for JIT compilers unaware of the special naming convention. We rely on the naming convention defined by Mono in this library. On the x86 platform, Mono's JIT compiler recognizes the special semantics. Many other platforms are supported by Mono, but the SIMD extensions are not implemented yet.

4 Experiments and Analysis

This section presents our experiments with some loop kernels. It shows how initial results were far from acceptable, and it analyzes what are the minimum conditions to produce efficient code.

4.1 Setup

Our goal is to illustrate the advantage of vectorizing *bytecode*. The focus in on the specificities of target independent bytecode. Effectiveness of vectorization as an optimization technique is *not* the point of our work, it has been proved already elsewhere, and we take it for granted, as any other optimization mentioned in this paper. For this reason, we decided to show some results only on small kernels that illustrate the *key features* of vectorization. Using real application would only blur the specific behaviors we are interested in. We use the same benchmarks as [17]. See Table 1 for a short description. They cover several data types and type sizes (single precision floating point, 8-bit and 16-bit integers). They also illustrate various features of the vectorizer: simple arithmetic, reductions, use of a constant. All kernels operate on arrays of 1000 elements, except *sum_u8* and *sum_u16* which operate on 10,000 elements to keep the running time in the order of a few seconds. Each kernel is also wrapped by a main loop that executes many times.

Since those benchmarks, from the BLAS suite, are written in Fortran, we wrote a straightforward implementation in C. We used the latest release of Mono at the time of writing: version 2.4.2.3. The CLI backend is based on GCC version 4.4. In order to demonstrate both the performance advantage and the portability, we run our experiments on several hardware platforms:

- a desktop PC featuring an Intel Core2 Duo clocked at 3 GHz — supporting the SIMD extensions MMX, SSE, SSE2, SSSE3 — running Linux 2.6.27;
- a Sun Blade 100 featuring a TI UltraSparc IIe, clocked at 500 MHz, running Linux 2.6.26.

Name	Description	Data type	Features
vecadd_fp	addition of two vectors	floating point	arithmetic
sdot_fp	dot product of two vectors	floating point	reduction
saxpy_fp	constant times a vector plus a vector	floating point	constant
dscal_fp	scale a vector by a constant	floating point	constant
max_u8	find maximum over elements of a vector	8-bit char	reduction
sum_u8	sum the elements of a vector	8-bit char	reduction
sum_u16	sum the elements of a vector	16-bit short	reduction

Table 1. Description of the benchmarks

Note that, even though the UltraSparc has a SIMD instruction set extension VIS, the Mono JIT compiler does not exploit these extensions yet. It does support the SSE extension on the x86 architecture. We also simulate an x86 platform without SIMD support by running the experiments on the same desktop PC and by disabling the SIMD intrinsics recognition.

4.2 Initial Results

In this first experiment, we run the benchmarks with three configurations on the x86 platform. In all cases of Table 2, the C programs are compiled to bytecode, and the bytecode is run by the Mono JIT compiler.

1. Firstly, we generate the bytecode without the vectorizer. That is, we use GCC with the command line flags `-O2 -fno-tree-vectorize` (note that, as of today, `-O2` would be sufficient since the vectorizer is only enabled at `-O3`). This gives us our baseline, reported in the column *scalar* of Table 2.
2. Secondly, we compile the benchmarks again, with the vectorizer enabled, using `-O2 -ftree-vectorize`. Running times are reported in the columns *vectorized* as absolute values and as performance relative to the scalar version (defined as the scalar base time divided by the new time). The next column, labeled *max* indicates the rough maximum speedup one would naively expect, based on the data type.
3. Finally, the vectorized bytecode produced in the previous step in run again, but without SIMD support in the JIT compiler. On x86, this is achieved by adding the flag `--optimize=-simd` to Mono.

At first glance, we can make the following high-level comments:

- *sdot* is not vectorized. This is confirmed by the activating the vectorizer's debugging messages. Figuring out the cause of this missed optimization was beyond the scope of this paper, and we omitted *sdot* in the rest of this paper.
- The vectorized bytecode shows significant speedups, ranging from 1.8 to 9.1.
- Even though the speedups are high, one might expect even more: 32-bit floating point values packed in 128-bit vectors gives an upper bound value for the speedup of 4 for the benchmarks *vecadd*, *saxpy*, *dscal*. The kernels *max_u8* and *sum_u8* operate on 8-bit integers, let us expect a 16x speedup. *max_u8* achieves a reasonable score of 9.1, but *sum_u8* obviously has a problem.
- The performance of the vectorized code run without SIMD support is unacceptable, with relative performance in the range 0.22 to 0.52.

benchmark	scalar	vectorized			no SIMD	
	time	time	rel. perf.	max	time	rel. perf.
vecadd_fp	1893	815	2.3	4	4475	0.42
sdot_fp	3039	3039	1.0	4	3039	1.0
saxpy_fp	2429	1224	2.0	4	8531	0.28
dscal_fp	1921	733	2.6	4	4122	0.47
max_u8	3156	346	9.1	16	6101	0.52
sum_u8	9030	2613	3.5	16	32528	0.28
sum_u16	9334	5223	1.8	8	42339	0.22

Table 2. Initial performance results on x86 (10^6 iterations, time in ms)

benchmark	scalar	previous	no SIMD	
	time	rel. perf.	time	rel. perf.
max_u8	3156	0.52	5185	0.61
sum_u8	9030	0.28	20569	0.44
sum_u16	9334	0.22	27565	0.34

Table 3. Performance results on x86, with inlining

4.3 Missed Inlining

We first look at the disastrous performance of the code in the absence of SIMD support, especially the bottom two kernels of Table 1. It turns out that the function which implements the vector operation (sum or max) is not inlined as we expected. It is inlined in the case of the floating point kernels. We found out that the coding style in Mono.Simd differs according to the size of the vector: 4-element vectors arithmetic (e.g. floating point) is implemented as four sequential statements, while 8-element and more vectors are implemented with a loop. This loop changes the outcome of the inlining heuristic of the JIT compiler. We rewrote the code of the library and reran those three tests, obtaining the numbers of Figure 3.

Unfortunately, the call to max is still not inlined, but we still obtain slightly better code for *max_u8*. Even though the slowdown of the *sum* kernels is still unacceptable, the improvement is also noticeable.

4.4 Manual Improvements

We then take advantage of the tracing capabilities of the JIT compiler and we dump the code emitted for *vecadd* for manual inspection. Functions calls are effectively inlined, however the resulting x86 code looks extremely poor. The loop contains 91 instructions. In comparison, the emitted code for the scalar version is 12 instructions long. Since combining vectorization and inlining amounts to unrolling the loop four times, one would expect in the order of 48 instructions, not taking into account induction variable simplification. Careful analysis shows that the input values (the vector elements) are copied twice in the function stack frame before being loaded in the floating point unit, and the output values are copied three times on their way from the floating point unit to their final destination. Our hypothesis is that the JIT compiler is missing a simple pass of copy propagation and dead code elimination after inlining.

benchmark	scalar	vectorized			no SIMD	
	time	time	rel. perf.	max	time	rel. perf.
vecadd_fp	1197	537	2.2	4	1228	1.0
saxpy_fp	1544	724	2.1	4	1890	1.2
dscal_fp	1045	657	1.6	4	1095	1.0
max_u8	3541	227	15.6	16	3735	1.1
sum_u8	6707	1277	5.3	16	8925	1.3
sum_u16	6710	2547	2.6	8	8198	1.2

Table 4. Final results on x86

We manually optimize the code produced by Mono, and link it again with the loop driver in the main program[1]. Since the loop is a single basic block, copy propagation and dead code elimination are straightforward, and very much within the capabilities of a JIT compiler. Running the experiment again, we measure 1228 ms, which is 3.6 times faster that the original. We apply the same simple cleanup to all vectorized benchmarks, we obtained improvements ranging from 3.6 to 5.2, with the exception of *max_u8* which scores "only" 1.6 because of the presence of the function calls.

In the interest of fairness, we apply the same simple manual code optimizations on all the generated versions of the loops of Table 2: scalar, vectorized, and vectorized without SIMD support. The consolidated results are presented in Table 4.

4.5 UltraSparc

We run the same set of experiments on the UltraSparc workstation, with the exception that Mono does not implement the SIMD code generation for this processor yet. All the finding on the x86 architectures have their counterpart on the UltraSparc, which is not surprising since the JIT compiler engine is the same. Because of the RISC nature of the UltraSparc, loading a long immediate in a register requires two instructions, and the address computation of an array element requires a third instruction for the addition. Since the first two instructions load a constant address, they are loop invariant and it was an additional obvious manual optimization to hoist them outside the loop. Table 5 shows the final results.

Speedups come from the fact that inlining the vector operations is similar to unrolling the small loops. In the case of the reduction kernels, this speedup is offset by the reduction epilogue and the less-than-optimal code generation.

4.6 Final Comments

The optimizations we run manually are basic compiler optimizations, which run in linear time. They are by no means out of reach of a JIT compiler. It should simply be a matter of running them again before code emission.

The generated vectorized code can be further improved, beyond our manual cleanup. In particular, intermediate results are continuously loaded and stored on

[1] In doing this, we potentially bias our results by omitting the compile-time of the JIT compiler. We verified that this is not the case: Mono reports compiling time below 0.25 ms for each of our loop kernels.

benchmark	automatic			after manual cleanup		
	scalar	vectorized	rel. perf.	scalar	vectorized	rel. perf.
vecadd_fp	4215	11193	0.38	2810	1947	1.4
saxpy_fp	5216	21011	0.25	3812	3239	1.2
dscal_fp	3642	10687	0.34	2608	1787	1.5
max_u8	3151	8613	0.37	3032	3188	0.95
sum_u8	10022	65864	0.15	8019	8559	0.94
sum_u16	10756	109866	0.10	8788	11256	0.78

Table 5. Final performance on Sparc, 10^5 iterations, time in ms

the stack at each iteration. The scalar version manages to keep intermediate values in registers. Inductions variables are not optimized in the vectorized code as aggressively as in the scalar code. Those reasons explain why the floating point benchmarks do not reach a better speedup.

Reduction kernels are impacted by another problem. An epilogue performs the final reduction over the 8 or 16 elements of the vector result. This code is less optimized than the loop body, and it impacts to total performance. Adding new builtins for *horizontal* operations (like the sum of the elements of a vector) could help the JIT compiler generate better code even for the epilogue.

5 Related Work

Bytecode formats and JIT compilers have existed for decades. CLI has recently drawn a lot of attention. Many projects exist, both commercial and open source. Microsoft proposed the .NET [16] framework. Mono [13] has been presented in Section 3. ILDJIT [2] is a distributed JIT compiler for CLI. LLVM [11] is a compiler infrastructure that defines a virtual instruction set suitable for sophisticated transformation on object code.

The original work in the GCC auto-vectorizer has been presented by Nuzman and Henderson in [17, 18]. They showed how the vectorizer algorithms can be implemented in a target-independent way, and driven by the compiler machine model. In their approach, the generated code *is* target-dependent, while in ours even the generated code is target-independent. We postpone the specialization to the run-time.

Leśnicki et al. [12] proposed to annotate the CLI bytecode with information to help the run-time vectorizer. In that approach, the vectorization happens at run-time, but the static compiler does the analysis and hints at the interesting loops. However, marking all the relevant loops and variables with the appropriate and usable information, while keeping legal CLI code, proved difficult. In this paper, we propose to entirely apply the vectorization in the static compiler, and to possibly revert to scalar code when necessary.

El-Shobaky et al. [9] also propose to apply the vectorization at run-time. They unroll loops to duplicate loop bodies, and they modify the code selector of the JIT compiler to group corresponding instructions using tree pattern matching techniques. However only a small number of operators are supported, and the number of additional rules in the code selector will grow rapidly when more complex patterns

are needed. The approach also suffers a number of limitations, as described in the paper.

Vector LLVA [1] is similar to our approach, but for the LLVM instruction set. However, in contrast to our proposal, the authors do extend the bytecode itself, breaking the compatibility, and they do not investigate the behavior of vector code on non-SIMD machines.

Clark et al. propose to rely on a simple dynamic translation mechanism to recognize the instruction patterns that can be vectorized (those patterns have been produced by scalarizing the output of a vectorizing compiler). They are able to handle data widths increases. The solution, however, is entirely in hardware and thus limited in the size of the instruction window in which it can recognize patterns.

6 Conclusion

In this paper, we propose a scheme to reconcile platform neutral binary formats like bytecodes, and the careful exploitation of SIMD extensions of instruction sets.

Our contribution is three fold: first, we added vectorization capability to the GCC CLI backend and we made our developments publicly available. Second, we showed that vectorized bytecode is a viable approach to deliver performance in the presence of SIMD instructions while not incurring any penalty on non-SIMD instruction sets. And third, we analyzed under what conditions the bytecode produced by the static compiler can be efficiently executed on various processors. In particular, the JIT compiler must guarantee that some basic optimizations will be run in order to not to degrade the performance.

References

1. Robert L. Bocchino, Jr. and Vikram S. Adve. Vector LLVA: a Virtual Vector Instruction Set for Media Processing. In *VEE'06*, pages 46–56, June 2006.
2. Simone Campanoni, Giovanni Agosta, and Stefano Crespi Reghizzi. A parallel dynamic compiler for CIL bytecode. *SIGPLAN Not.*, 43(4):11–20, 2008.
3. Nathan Clark, Amir Hormati, Sami Yehia, Scott Mahlke, and Krisztian Flautner. Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping. In *HPCA'07*, pages 216–227, Washington, DC, USA, 2007.
4. Marco Cornero, Roberto Costa, Ricardo Fernández Pascual, Andrea Ornstein, and Erven Rohou. An experimental environment validating the suitability of CLI as an effective deployment format for embedded systems. In *Conference on HiPEAC*, pages 130–144, Göteborg, Sweden, January 2008. Springer.
5. Roberto Costa, Andrea C. Ornstein, and Erven Rohou. CLI back-end in GCC. In *GCC Developers' Summit*, pages 111–116, Ottawa, Canada, July 2007.
6. Roberto Costa and Erven Rohou. Comparing the size of .NET applications with native code. In *3rd Intl Conference on Hardware/software codesign and system synthesis*, pages 99–104, Jersey City, NJ, USA, September 2005. ACM.
7. Koen De Bosschere, Wayne Luk, Xavier Martorell, Nacho Navarro, Mike O'Boyle, Dionisios Pnevmatikatos, Alex Ramirez, Pascal Sainrat, André Seznec, Per Stenström, and Olivier Temam. *High-Performance Embedded Architecture and Compilation Roadmap*, volume 4050 of *LNCS*, pages 5–29. 2007.

8. ECMA International, Rue du Rhône 114, 1204 Geneva, Switzerland. *Common Language Infrastructure (CLI) Partitions I to IV*, 4th edition, June 2006.
9. Sara El-Shobaky, Ahmed El-Mahdy, and Ahmed El-Nahas. Automatic vectorization using dynamic compilation and tree pattern matching technique in Jikes RVM. In *ICOOOLPS '09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 63–69, New York, NY, USA, 2009. ACM.
10. Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, February 2008.
11. Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO'04*, Palo Alto, California, Mar 2004.
12. Piotr Leśnicki, Albert Cohen, Grigori Fursin, Marco Cornero, Andrea Ornstein, and Erven Rohou. Split compilation: an application to just-in-time vectorization. In *GREPS'07, in conjunction with PACT*, Braşov, Romania, September 2007.
13. The Mono Project. `http://www.mono-project.com`.
14. José M. Moya, Javier Rodríguez, Julio Martín, Juan Carlos Vallejo, Pedro Malagón, Álvaro Araujo, Juan-Mariano Goyeneche, Agustín Rubio, Elena Romero, Daniel Villanueva, Octavio Nieto-Taladriz, and Carlos A. López Barrio. SORU: A reconfigurable vector unit for adaptable embedded systems. In *ARC '09: Proceedings of the 5th Intl. Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, pages 255–260, 2009.
15. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
16. Microsoft .NET. `http://www.microsoft.com/NET`.
17. Dorit Nuzman and Richard Henderson. Multi-platform auto-vectorization. In *CGO'06*, pages 281–294, Washington, DC, USA, 2006. IEEE Computer Society.
18. Dorit Nuzman and Ayal Zaks. Autovectorization in GCC – two years later. In *GCC Developers' Summit*, pages 145–158, June 2006.
19. Erven Rohou. Combining processor virtualization and split compilation for heterogeneous multicore embedded systems. In *Emerging Uses and Paradigms for Dynamic Binary Translation*, number 08441 in Dagstuhl Seminar Proceedings.
20. Gabriele Svelto, Andrea Ornstein, and Erven Rohou. A stack-based internal representation for GCC. In *GROW'09*, pages 37–48, Paphos, Cyprus, 2009.

Compiler-controlled and Compiler-hinted Voltage Scaling Approaches

Dmitry Zhurikhin[1], Andrey Belevantsev[1], Kirill Batuzov[1], Valery Ignatiev[1], Roman Zhuykov[1], and Semun Lee[2]

[1] Institute for System Programming, Russian Academy of Sciences
{zhur,abel,batuzovk,rook,zhroma}@ispras.ru
[2] Samsung Corp.
semun.lee@samsung.com

Abstract. This paper reports on the two approaches to dynamic voltage and frequency scaling (DVS) hinted by GCC and controlled by Linux kernel. The first approach uses profiling information for marking DVS regions which should be executed with lower frequency, while the kernel does the actual switching on entry and exit of those regions, taking into account possible switching requests from multiple processes. In the second approaches, the kernel itself decides when and to what value the frequency would be switched, and the compiler provides simple information on behaviour of program regions. For both approaches, a light-weight sampling-based profiling technique is developed. Results show some CPU energy savings with both approaches, but not whole-system energy savings on the test boards used.

1 Introduction

In our previous work [14] we have evaluated some of the compiler techniques for lowering the CPU energy consumption using GCC compiler, including compiler-controlled voltage scaling (DVS), bit-switching minimization, and memory optimizations. The most promising technique was found to be DVS, which provided several per cent CPU energy savings in our testing and very small overall system energy savings. However, the initial implementation of our DVS technique had limitations, mainly a) being an intraprocedural transformation (and thus considering neither interprocedural program regions nor any regions containing function calls as candidates for DVS) and b) not taking the multiprocess environment into account.

We have conducted a research project that aimed at removing those limitations. We have developed two DVS algorithms. One is a fully static interprocedural DVS approach (that is, a compiler controls the points of changing frequency and the values on which it should be changed) that uses a kernel manager for handling conflicts of queries for frequency changes between different processes. The other DVS approach is implemented in the kernel as a `cpufreq` governor that uses compiler-provided information for making decisions on DVS (a so-called "mixed" approach). We have also developed a light-weight

sampling-based profiling mechanism that is used in both approaches for devising the needed information.

The rest of the paper is organized as follows. Section 2 describes the static DVS approach together with the devised profile support. Section 3 reports on the mixed DVS approach. Section 4 provides experimental results. Section 5 concludes.

2 Static Interprocedural DVS Operating in Multiprocess Environment

The developed static DVS algorithm is based on the algorithm in [3] and its implementation in our previous work [14]. We will shortly highlight the main stages of the algorithm for clarity. The basic idea of the algorithm is to divide a program into single-entry/single-exit (SESE) regions and to estimate their execution time on each frequency from profiling data. Then, given that we know CPU power consumption on each frequency from hardware specifications, and we know frequency switching latency from experiments, we can assign any execution frequency for each region and still we will be able to estimate the total time and energy needed to execute the program. Therefore, this data is enough to solve the optimization problem of finding the set of regions that provides the lowest energy consumption while meeting some deadline on execution time.

Our implementation in [14] operates on a single function at a time and consists of the following stages:

1. Construct basic regions and combined regions. A basic region is just a basic block or a loop, while a combined region is a SESE region made from basic regions.
2. For every basic region, profile overall execution time at each available processor frequency, $T(R, f)$, and the number of times a region is executed, $N(R)$.
3. Estimate $T(R, f)$ and $N(R)$ for combined regions[3].
4. Find the best region (basic or combined) and its execution frequency using which minimizes CPU energy consumption and does not increase running time above given threshold (controlled by user).
5. Insert frequency switching commands at the entry and the exit of the selected region.

Compared to [14], we have improved the following parts of the algorithm: building program regions suitable for DVS (more combined regions are considered suitable); finding the regions which will be executed with lower frequency (a set of regions is considered instead of a single region); profiling mechanism (a light-weight sampling-based profiling is used); and working in multiprocess environment (by handling queries for switching frequency from different processes in the kernel). We will expand on these improvements in the following subsections.

[3] $N(R)$ is taken from the basic region that is at the entry of the combined region R. $T(R, f)$ is computed as sum of $T(BR, f)$ over all basic regions BR that form the combined region R.

2.1 Building Program Regions Suitable for DVS

The original intraprocedural implementation did not allow us to handle regions with calls, such regions were not considered for optimization. In this paper, static DVS optimization is implemented as an interprocedural pass in GCC, so we have removed this restriction.

We build regions in two stages. The basic regions for each function are built during early local passes and stored in `struct function` (GCC per-function data). The reason for this is that the basic regions are actually profiled, so their construction should happen at the moment when profiling information is read. The actual optimization needs to see regions from all functions, so it happens in an interprocedural pass later in the compiler pipeline.

In this pass, the combined regions are built for each function as SESE regions (i.e. region body should be dominated by entry and postdominated by exit of the region) whose blocks all belong to previously constructed basic regions. These basic regions are allowed to contain calls. A combined region still may not cross function boundaries though, as this was not implemented. However, combined regions from all functions are merged in a single array, so that the solving part of the DVS optimization can consider all regions at once.

It can be noted that constructing basic regions at one point of the compiler pipeline and using them for optimization at another point creates the problem of keeping the regions consistent between the two passes. This problem is solved similarly to the problem of updating profile information: when control flow is modified through the GCC `cfghooks.c` API, that is, when a basic block is created, deleted, added to or removed from a loop, or merged with another block, the information about basic region is updated accordingly. We have also written a verifier to check the region consistency that is called when control flow is modified.

2.2 Finding The Best Set of Regions to Perform DVS

We choose the set of regions on which we need to change frequency in an interprocedural pass, after constructing combined regions of all functions in a translation unit. The problem we solve is as follows. We consider two available frequencies for program execution, maximum and minimum. Let us consider T as the extra time we can use for program execution (calculated as $p\%$ slowdown on execution time using the maximum frequency) and n program regions. Each region has weight w_r, calculated as the extra time needed to execute the program using the minimum frequency and the latency of switching frequency, and value v_r, calculated as the energy saved when the region is executed on the minimum frequency[4]. Now the problem of choosing the optimal set of regions can be formulated as a 0-1 knapsack problem. However, there is additional restriction on regions when operating in interprocedural mode: some regions may intersect with each other (e.g., a region containing a call and a region in the callee), and the regions we are choosing may not intersect.

[4] We only consider regions with $v_r > 0$ and $w_r < T$.

We have implemented two algorithms for solving our problem. The first algorithm is a simple greedy algorithm. We sort the regions based on v_r/w_r ratio and we choose regions starting from the one with the lowest ratio. We add a region to the set when it does not intersect with the regions that are already in the set and when their total weight does not exceed T. If a region does not meet these conditions, we consider the region with the next best ratio. The final set is found when we process all regions.

The second algorithm is a backtracking algorithm that is used in case of small number of regions to find the optimum solution. We sort the regions in ascending order by weights and then in descending order by values, and we assign indexes to the regions according to the resulting order. On each step, we try to add the next region to the candidate set considering only regions with indexes greater than the ones already in the set. When the current region intersects with some of the candidate set regions, we process the next region in the sort order. When adding the current region to the set the total set weight will exceed T, we don't process next regions as due to the sort order they will be too costly. In this case, and also when the last region is processed, we backtrack by removing the region with the largest index from the set and proceeding with adding the next region. When we have succeeded in adding the region, we remember the current best solution and proceed to the next region. In addition, we prune the search space by backtracking immediately when adding all remaining regions to the set will not provide better solution than the current best solution.

We can also model the intersecting regions by considering the graph G whose vertexes correspond to program regions and whose edges connect a pair of regions which do not intersect. Then a set of regions eligible for switching frequency will form a clique in G, and our goal will be to find a clique whose vertices have the maximum sum of values while having the sum of weights not greater than T. One of the possible solutions to this problem can be found in [5]. We did not implement this approach, as it is not obvious it would provide much better solutions for our tasks.

2.3 Handling Recursive Functions

We need to make some additional efforts to handle regions that contain calls to recursive functions. As the call graph may be incomplete, and it also may have indirect calls, we may fail to detect recursive calls (i.e., loops in the call graph) during compilation time. To solve this issue, we have created wrappers around instructions for changing frequency, which are added to `libgcc`. For each region, we compute its hash based on the source file name, the function name, and the region number. When calling the wrapper, we store the region hash and the call depth. The frequency is lowered when the depth equals to zero, then with each subsequent request having the region hash equal to the stored one the depth is increased. The frequency is raised back when the depth equals to one and the region hash equals to the stored one.

2.4 A Sampling-based Profiler

The initial profile mechanism for DVS implemented in [14] turned out to be too heavyweight, so the DVS optimization basically worked using incorrect data. We have developed a lighter profiler based on kernel timer interrupts. The instrumented program and the kernel communicate via shared memory. When the program starts, it requests a shared memory region from the kernel via the `ioctl` call. The shared memory holds a stack of currently executing DVS regions. When entering region R, its ID is pushed to the stack. When leaving R, region IDs are popped from the stack until the ID of R is removed. This allows gathering correct statistics in case we haven't tracked some region exits. When a timer interrupt occurs, all region counters that are currently stored on the stack are updated. When the program is finished, the resulting sample data is written on disk so that GCC can parse it later.

There is a problem of constructing a proper region ID. As we noted, a region is identified by a source file name, a function name, and a region number. We would like to fit the ID in a 32-bit number. However, as we need some space for the function name and the region number, it would be hard to hash the file names so that the hash function values will fit in the remaining space, so with a large program (several hundred files) the collisions in detecting the regions could very probably happen. To avoid this situation, we use a counter of compiled translation units as a hash, stored in a separate file and incremented once per compilation. Of course, this cannot be used in a production GCC, as this leads to differences in code generation of the same file compiled several times.

Last thing to note here is that the profiling info for basic regions is updated together with them between the early local pass of constructing basic regions/reading profile information and the interprocedural pass of performing the DVS optimization.

2.5 Handling DVS Requests from Multiple Processes

In a multiprocess environment, it is possible that the requests on frequency change from several DVS processes will conflict. For this case, we have implemented the mechanism of changing frequency as a modification of `ondemand` governor of `cpufreq` instead of directly calling the wrappers added to `libgcc`. The `ondemand` governor measures the periods of idle CPU and the periods of executing useful code. When the ratio of these values exceeds certain threshold, the CPU frequency is raised; when it is below another threshold, the frequency is lowered, otherwise it is left unchanged.

We have modified the `ondemand` governor so that when a DVS program executes the region that should work on the lower frequency, the manager considers its execution time as idle CPU, so that the decision of lowering frequency becomes more probable. The communication between the program and the kernel happens through shared memory similarly to the profiling mechanism. The choice of communicating with the kernel or using direct frequency changing calls (i.e. pure static DVS) is controlled by a GCC option.

3 A Compiler-hinted Mixed DVS Approach

A mixed DVS approach is based on the idea that the CPU frequency control should be done in an OS kernel dynamically, while using information about program execution that can be gathered statically by a compiler. We have begun our work by studying a number of state-in-the-art mixed DVS approaches. Since most of these approaches are aimed at real-time systems, we couldn't follow them exactly as we don't expect to have so much information about the running application. Instead we have decided to enhance one of the pure online algorithms with using additional information from the compiler.

The following subsections will provide more details about the related work in mixed DVS approaches and Linux power management, our Linux kernel algorithm, and our compiler algorithm.

3.1 Related Work on Mixed DVS

We will present two mixed DVS approaches that are most interesting to us. The approach proposed by Azevedo et al. [1] is based around *checkpoints*. A checkpoint is a special place in the program's code marked with a label. It serves as a point where the calculations of the needed CPU frequency are done. During program compilation the program time constraints are set for the execution of the code regions between checkpoints, in terms of acceptable lower and upper bounds. Such information is stored in a special *checkpoint database*, along with the possible checkpoint transitions derived from the program control flow. Similarly to our approach, the program is profiled and run in order to get representative data on its power consumption.

The actual CPU frequency scaling takes place during program execution and can be done either by OS or by the code inserted at the checkpoints by the compiler. The main idea of the scaling phase is to generate at each checkpoint a list of *events*[5] that may arise later when executing the program. Based on this list, the upper frequency bound[6] and the optimum frequency[7] are computed, and the frequency found is set accordingly. The drawbacks of this approach are that it doesn't take into account the additional power and time that are needed to calculate the new CPU frequency and to set it, and that the approach was only tested on a simulator. At the same time, the algorithm in [1] provides the flexible way of specifying desired time and power properties of the compiled program.

The other approach by AbouGhazaleh et al. [2] is similarly divided into two stages, offline and run-time. Initially, the data on program execution is collected during profiling. At the offline stage this data is used to compute when

[5] An event contains a list of next possible checkpoints.

[6] The bound shows the maximum CPU frequency that should be set in order to save any power.

[7] The optimal frequency is the CPU frequency that should be set in order to satisfy time constraints of all next possible checkpoints (i.e., the event list).

and how frequently the *power management points*[8] (PMPs) will be called. Also, during the offline phase the program is instrumented with *power management hints*[9] (PMHs). Each PMH consists of the compiler inserted code that computes the worst-case remaining cycles starting from the current PMH location to the program end. This value may vary dynamically based on the executed path for each run. For example, the remaining cycles at a PMH inside the function body are dependent on the path from which the function is called.

During run-time, a PMH computes and passes dynamic timing information to the OS in a predetermined memory location which holds the most recent value of the estimated worst-case remaining cycles. Periodically, a timer interrupt invokes the OS to execute the PMP code, so the OS adjusts the CPU frequency based on the latest value and the remaining time to the program deadline. The drawback of this approach, common with the previous one, is that they are aimed at real-time systems, so they could not be used directly as such.

3.2 Power Management in Linux

There are three widespread approaches to Linux power management:

- implemented as a stand-alone application or a daemon, e.g. CPUSpeed [6] or Open Hardware Monitor [7] projects;
- implemented as a stand-alone module or a patch for the kernel, e.g. the Dynamic Power Management project [8] or the approaches based on pre-2.6 Linux kernels;
- implemented as a governor of `cpufreq`, which is the Linux kernel module that controls the CPU frequency added in the 2.6 kernel version.

At present, the Linux kernel contains five simple CPU frequency governors: `userspace`, which allows the user to set any desired supported frequency; `powersave`, which automatically sets minimum supported CPU frequency; `performance`, which sets maximum supported CPU frequency; `ondemand`, which is an interval-based dynamic CPU frequency scheduler[10], increasing the CPU frequency when the calculated load is more than 80% and decreasing it when the load is less than 20%; and `conservative`, which is very close to the `ondemand` governor, but it makes its decision also looking at the load of previous intervals. All these governors are implemented as modules and depend on the `cpufreq` module. This allows loading/unloading the governor modules and switching between them at runtime.

There are three canonical DVS algorithms proposed in [10], which are OPT, FUTURE, and PAST. The first two are impractical as they are able to look into

[8] A power management point shows the moments of time when the OS makes a decision on the new CPU frequency.

[9] The hints allow the OS to estimate the time remained until program ends; they are used by the CPU frequency manager.

[10] The `ondemand` governor calculates the CPU load on the last time interval as the sum of the run times of all tasks from the last interval divided by the interval length.

the future of the trace data and are used for reference purposes only. The latter is a practical variant formulated as a result of experiments with the former two. OPT is an unbounded-delay perfect-future algorithm that uses available energy in an optimal way by stretching the run times in a trace to fill all available idle time. While the algorithm is simple, it is unfeasible as it doesn't care when a specific job completes as long as it does so before the end of the total time span of the trace. As a result, OPT can produce large delays in jobs' run time and cannot give adequate response to real-time events. FUTURE is a modification of OPT that can only look into the future by a small window of the next allocated time interval. Energy consumption is optimized within the window while making sure no work is delayed past the end of the window. FUTURE approaches OPT in terms of energy savings for large windows, while for small ones its energy savings are small as well. The other advantage of this algorithm is that no response is delayed past the end of the window giving good real-time response in the case of small window sizes. The PAST algorithm uses a window in the past instead of looking into the future. PAST assumes that the workload in the next window will be the same as the previous one. As with FUTURE, the window size can be adjusted to give different performance results. Its performance has subsequently been evaluated as relatively good even compared to newer and more sophisticated algorithms [9].

The AVGn algorithm by Pering et al. [11] computes an exponential moving average of the previous windows. Again, the idea is that the workload of the next time interval is expected to be similar to the previous ones. AVGn improves on the three similar algorithms that predict workload by searching for patterns in the past CPU utilization, hoping that more intelligent heuristics will lead to larger energy savings. The CYCLE algorithm is based on the idea that the CPU utilization may be structured in a cyclical pattern of interleaved peaks and valleys, a phenomenon that is observed often on CPU utilization graphs. When the algorithm finds such cycles in the past intervals, it sets the CPU speed such that the amount of work needed with any excess cycles still left can be completed within the next window. When no pattern can be found, the load is predicted to be constant[11]. PATTERN is a generalization of the CYCLE method meant to detect any kind of pattern in the load level data of the previous intervals. When a match is found, the load of the next interval is predicted to be the same as that of the interval following the previous occurrence of the sequence. Finally, the PEAK algorithm looks in the past for high peaks of activity interspersed with more stable plains. Its prediction uses several heuristics based on the expectation of narrow peaks.

The other more advanced approaches for controlling the CPU frequency are usually aimed at special types of system load, e.g. when practically just media applications are run. Such approaches present good results on corresponding workflows, hence it is useful to know which types of load and kinds of applications are expected for the target system.

[11] The constant is given as a parameter to the algorithm.

Surprisingly, performance results of the described algorithms do not indicate that the more complex heuristics outperform the simpler algorithms. According to simulations in [9] and [12], the AVGn algorithm with its simple averaging performs best of all, while CYCLE, PATTERN, and PEAK approaches are only slightly better than PAST. Most of the described algorithms are implemented and evaluated for the Linux kernel during the project YADDA [13].

3.3 Linux Kernel Part of Our Mixed DVS Approach

We have implemented our mixed DVS approach as a new `cpufreq` governor called `mixeddvs` using performance measurement counters similarly to [4]. When the governor is active, programs compiled with the support of this mixed DVS will provide it with their status to guide its decisions. When such program is executed, first it communicates with the governor via the `ioctl` call. The kernel then allocates a hunk of shared memory for the program and registers its process in the list of so-called *controlled* tasks, also starting the hardware counters if this is the first controlled process.

The majority of the work of the kernel manager is done at the periods of process scheduling, when the previous process (which just left the CPU) and the next process (which is going to be executed) are known. If the previous process is the controlled one, the kernel reads the hardware counters' values and the shared memory values. The hardware counters' events are `CCNT`[12], `INSTR`[13], and `DCM`[14]. The shared memory value is $\texttt{REG}_{\texttt{TPI}}$, an average time for executing a single instruction on the maximum CPU frequency. It is profiled at the compilation stage and then is written to the shared memory via the instrumentation code.

Based on these values and on available slack time for slowdown, the kernel manager selects the new CPU frequency to be used when the previous controlled process will be executed next time. First, the slack time is modified as follows: *slack-time* `-=` $\texttt{CCNT}/\texttt{F}_{\texttt{cur}} - \texttt{INSTR} \cdot \texttt{REG}_{\texttt{TPI}}/\texttt{F}_{\texttt{max}} \cdot (100 + \texttt{pf}_{\texttt{loss}})/100$, where $\texttt{F}_{\texttt{cur}}$ is the current CPU frequency, $\texttt{F}_{\texttt{max}}$ is the maximum CPU frequency, and $\texttt{pf}_{\texttt{loss}}$ is the percent of allowed slowdown. The first part of the right-hand side of the formula shows the CPU time actually spent by a program, while the second part shows the CPU time the program would have spent (on the maximum CPU frequency), also increased by the slowdown percent. The difference shows the CPU time[15] used by the task that is above the slowdown percent. When the resulting slack time is too low or too high, the manager selects the maximum or the minimum CPU frequency accordingly, otherwise it looks at the hardware counters' values to find whether the currently executing region is good for DVS or not. We have tried several heuristics on this step, and we have found out that the relative number of data cache misses, `DCM`, works good. So, the lower or the higher CPU frequency is selected when `DCM` is high or low accordingly.

[12] A number of CPU cycles passed from the last counter reset.
[13] A number of ARM instructions executed from the last counter reset.
[14] A number of data cache misses.
[15] This time is positive when the process has used lower CPU frequency previously and negative otherwise.

When the new CPU frequency value is found, it will be used next time when the previous controlled process will be executed. The maximum CPU frequency is used for the previous process otherwise and also for the next process when it is not controlled. The only exception is when the process execution time is low, which is usual for daemons or short processes. The current CPU frequency will be kept then so that the number of the CPU frequency transitions will be lower. When the controlled process finishes or otherwise ends unexpectedly, it signals the kernel (again via `ioctl`) to free its data and the allocated shared memory.

The performance slowdown threshold is controlled in several ways. When selected, the `mixeddvs` governor creates a file called `performance_loss` in the `cpufreq` root directory. The value stored in the file represents the slowdown threshold that the new processes will have. It is also possible to set this value for the given program during compilation via the special option and the parameter.

3.4 GCC Part of Our Mixed DVS Approach

The GCC compiler part we used for the mixed DVS approach implementation is based on our previous work in Section 2. The profiling workflow is practically the same except that the single run on the maximum CPU frequency is needed for gathering data. This information includes $\texttt{REG}_{\texttt{TPI}}$ values for each region. The region $\texttt{REG}_{\texttt{TPI}}$ is low when it is CPU bounded and no memory stalls are present; when $\texttt{REG}_{\texttt{TPI}}$ is relatively high, then large number of memory stalls were encountered during region execution, so it is probably efficient to lower the CPU frequency on the region. $\texttt{REG}_{\texttt{TPI}}$ values are calculated by the kernel using `CCNT` and `INSTR` events pretty much the same way as with sampling-based profiling described in Section 2.4.

When the profiling results are used on the second compilation, GCC inserts the instrumentation code into the program, including the `ioctl` calls to allocate/free the kernel shared memory and the storing of $\texttt{REG}_{\texttt{TPI}}$ values of the DVS regions to the shared memory. As it is hard to predict the time of the next process scheduling, and as several DVS regions could be executed since the last process scheduling, the compiler also inserts code that averages $\texttt{REG}_{\texttt{TPI}}$'s over all executed DVS regions from the previous process scheduling.

4 Experimental Results

We have implemented both our approaches in GCC version 4.3.1 and Linux kernel version 2.6.24. We have evaluated our DVS optimizations (both static and mixed) together with the common `cpufreq` governors on the Aburto test suite, which represents many common scientific applications. The following measurement scheme was used. The CPU governor was set to the needed one before the test was run and then the `powersave` governor was set right after the test finish. Measurements were done during the same equal time for each run of the test. The average static power consumption of the test board was subtracted from the resulting power meter value.

Test results are shown in Table 1. Here, `time` is the actual run time of the test and is shown in seconds; `energy` is the CPU energy consumed during the measurement and is shown in mWh; the percent values show the difference between the given value and the performance column value. `Performance`, `powersave`, `ondemand`, `mixed`, and `offline` columns represent the runs with the maximum/minimum CPU frequency set, the kernel DVS approach, and our mixed and offline (with kernel manager) DVS approaches accordingly. The last two approaches had a slowdown threshold of 20%. Only some of the Aburto test programs are shown; other tests showed no significant differences in their run-time behavior.

		`performance`		`powersave`		`ondemand`		`offline`		`mixed`	
		time	energy	time	energy	time	energy	time	energy	time	energy
heapsort	val	63	6.7	87	4.3	63	7	67	6.7	69	5.8
	%	0	0	-38.1	35.82	0	-4.48	-6.35	0	-9.52	13.43
nsieve	val	56	5.8	65	3.3	58	5.8	66	3.4	59	5.1
	%	0	0	-16.07	43.1	-3.57	0	-17.86	41.38	-5.36	12.07
sim	val	132	15.1	219	9.7	133	15.4	212	14.3	180	12.8
	%	0	0	-65.91	35.76	-0.76	-1.99	-60.61	5.3	-36.36	15.23
tfftdp	val	82	10.12	132	6.77	82	10.16	82	10.15	97	9.42
	%	0	0	-60.98	33.1	0	-0.4	0	-0.3	-18.29	6.92
whets	val	164	21.1	323	15.2	164	21.7	194	19.5	202	20.4
	%	0	0	-96.95	27.96	0	-2.84	-18.29	7.58	-23.17	3.32
madmp3	val	169	9.2	169	7.7	169	7.7	169	7.4	169	7.5
	%	0	0	0	16.3	0	16.3	0	19.57	0	18.48
Total	val	666	68.02	995	46.97	669	67.76	790	61.45	776	61.02
	%	0	0	-49.4	30.95	-0.45	0.38	-18.62	9.66	-16.52	10.29

Table 1. Evaluation results.

The results show that the mixed DVS approach is able to save 10.3% CPU energy on average while slowing down execution on 16.5% in average, while the offline DVS approach is able to save 9.7% CPU energy at the cost of 18.6% slowdown.

5 Conclusions

We have completed our work by improving the static DVS approach over the one proposed in [14] by making it an interprocedural pass in GCC and via implementing a light-weight sampling-based profiling for receiving its data; and by making it interprocess via implementing a kernel manager for handling its requests. We have also implemented the mixed DVS approach as the new `cpufreq` governor using the data gathered via the similar sampling-based profiling and the shared memory for communication between the program and the Linux kernel.

We have evaluated both static and mixed DVS approaches. The results show that the mixed DVS approach is able to save 10.3% CPU energy on average while slowing down execution on 16.5% on average, while the offline DVS approach is able to save 9.7% energy at the cost of 18.6% slowdown. These results show some CPU energy consumption reduction, but not the whole-system energy consumption reduction. We believe that either the approaches should be evaluated on the real mobile devices, where the ratio of the CPU energy consumption to the whole system energy consumption will be larger, or they will be useful for the future devices with multicore CPUs that will again have the above ratio increased compared to the current one.

References

1. A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Profile-Based Dynamic Voltage Scheduling Using Program Checkpoints. In *Proceedings of the Conference on Design, Automation and Test in Europe*, March 2002, IEEE Computing Society.
2. N. AbouGhazaleh, D. Moss, B. R. Childers, and R. Melhem. Collaborative Operating System and Compiler Power Management for Real-Time Applications. In *ACM Trans. Embed. Comput. Syst.*, vol 5, 1, Feb. 2006, pp. 82-115.
3. C. Hsu. Compiler-Directed Dynamic Voltage and Frequency Scaling for CPU Power and Energy Reduction. Doctoral Thesis, Rutgers University, 2003.
4. K. Choi, R. Soma, and M. Pedram. Fine-Grained Dynamic Voltage and Frequency Scaling for Precise Energy and Performance Trade-Off Based on the Ratio of Off-Chip Access to On-Chip Computation Times. In *Proceedings of the Conference on Design, Automation and Test in Europe*, Volume 1, February 2004.
5. A. Massaro, M. Pelillo, and I. M. Bomze. A Complementary Pivoting Approach to the Maximum Weight Clique Problem. SIAM J. on Optimization 12, 4 (Apr. 2002), pp. 928-948.
6. CPUSpeed kernel module. `http://www.carlthompson.net/Software/CPUSpeed`
7. Open Hardware Module. `http://ohm.freedesktop.org`
8. Dynamic Power. `http://dynamicpower.sourceforge.net`
9. D. Grunwald, C. B. Morrey, P. Levis, M. Neufeld, and K. I. Farkas. Policies for Dynamic Clock Scheduling. In *Proceedings of the 4th Conference on Symposium on Operating System Design and Implementation*, Volume 4, San Diego, California, October 2000.
10. M. Weiser, B. Welch, A. Demers, and S. Shenker, S. Scheduling for Reduced CPU Energy. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, Monterey, California, November, 1994.
11. T. Pering, T. Burd, and R. Brodersen. The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms. In *ISLPED '98: Proceedings of the 1998 international symposium on Low power electronics and design*, pp. 76-81, 1998.
12. K. Govil, E. Chan, and H. Wasserman. Comparing Algorithms for Dynamic Speed-setting of a Low-power CPU. In *Mobile Computing and Networking*, pp.13-25, 1995.
13. The YADDA project. `http://www.eecg.toronto.edu/~tamda/csc2228`
14. D. Zhurikhin, A. Belevantsev, A. Avetisyan, K. Batuzov, and S. Lee. Evaluating power-aware optimizations within GCC compiler. Presented on GROW'09 workshop, January 2009, `http://www.doc.ic.ac.uk/~phjk/GROW09/papers/06-PowerBelevantsev.pdf`.

Using Software Metrics to Evaluate Static Single Assignment Form in GCC

Jeremy Singer[1], Christos Tjortjis[2,3], and Martin Ward[4]

[1] University of Manchester, UK
jsinger@cs.man.ac.uk
[2] University of Ioannina, Greece
[3] University of Western Macedonia, Greece
[4] De Montfort University, UK

Abstract. Over the past 20 years, static single assignment form (SSA) has risen to become the compiler intermediate representation of choice. Compiler developers cite many qualitative reasons for choosing SSA. However in this study, we present clear *quantitative* benefits of SSA, by applying several standard software metrics to compiler intermediate code in both SSA and non-SSA forms. The average complexity reduction achieved by using SSA in the GCC compiler is between 32% and 60% according to our software metrics, over a set of standard SPEC benchmarks.

1 Introduction

Static single assignment form (SSA) is a popular compiler IR. Since the formulation of SSA in the late 1980s [9] it has rapidly become the de factor standard IR for code analysis and optimization.

In terms of research compiler infrastructures, SUIF [17], Microsoft's Bartok [11] and Phoenix, IBM's Jikes RVM [6], and LLVM all use SSA-based intermediate forms. Several commercial compilers have recently been released as open-source. Among these, Java HotSpot (originally from Sun) and Open64 (originally from SGI) use SSA. Although Intel's C compiler is not open-source, it has been reported to make heavy use of SSA-based optimizations [34]. Regarding the open-source GNU Compiler Collection (GCC), its development began before SSA was well-characterized or widely known. However SSA support has been backported into the original optimization infrastructure [31, 32], as of version 4.0. Thus we see that many compilers, from a wide variety of vendors, for a diverse range of source programming languages and target architectures, use SSA.

Despite its popularity and many anecdotal success stories, there has been little previous work on formal evaluation of the reasons for SSA's advantages in static analysis and optimization. This paper employs software metrics to provide a *quantitative* evaluation of the comparative merits of SSA. We aim to make our observations as general as possible, by avoiding any restrictions or assumptions that are specific for systems, analyses or optimizations.

The major contribution of this paper is in the previously unchartered area of *program meta-analysis*, which concerns the analysis of properties of program analysis tools and techniques. The paper provides a *quantitative assessment* of the benefits of SSA in an analysis-independent manner, by means of *software metrics*. It shows with several standard measures that an SSA representation of a procedure has reduced complexity in relation to a non-SSA representation of the same procedure, in GCC. The average *complexity reduction* is between **32% and 60%** for selected complexity metrics, over a set of representative benchmark programs.

2 Program Representations

A control flow graph (CFG) is a static representation of possible flows of execution in a procedure. The CFG form is the basis for almost all classical (pre-SSA) intra-procedural data flow analysis techniques. For full details, consult compiler textbooks [1] [28] [30].

The conversion from CFG to SSA involves *systematic renaming* of variables. The key property of SSA is that each variable in the program must have a unique (hence *single*) definition point (hence *assignment*) in the program text (hence *static*). In order to achieve this, it is necessary to generate new variable names at definition points, and propagate these new names to all uses reached by that definition. At control-flow merge points reached by several definitions of a variable, it may be necessary to insert pseudo-assignments (known as ϕ-functions) to merge explicitly these renamed variable definitions into a new variable. In SSA form, each unique variable definition must dominate all uses of that variable.

SSA is perceived to improve data flow analysis, since: (1) it decreases the size of def-use chains; and (2) it enables finer-grained variable-specific data flow information.

A *def-use* chain links a variable definition with all its uses. It is an additional data structure alongside a CFG, encapsulating variable data flow in the program. For instance, if there are 10 definitions of variable x which can reach a control flow merge point, followed by 10 uses of x, then every definition can reach every use. This means there are 100 data flow links for x. In the SSA version of the program, there are just 20 data flow links: 10 from the definitions to the ϕ-function at the merge point, and 10 from the ϕ-function to the uses. Thus SSA can be seen to simplify def-use chains. Many authors extol this virtue of SSA and its consequent effect on splitting live ranges [10, 5, 19, 29, 4].

SSA enables effective sparse data flow analysis: Data flow information can be associated with a variable globally, rather than at each specific control flow point in the program [38]. Hasti and Horwitz assert that SSA encodes control flow information directly into variable names, which means that flow-insensitive analysis of SSA is as accurate as flow-sensitive analysis in certain cases [13].

Thus it is commonly recognised that SSA gives some advantages over the CFG. Research papers on SSA generally provide comparative evaluations of CFG *versus* SSA for a single data flow problem, compiler optimization or system;

for example [2]. However as far as we are aware, the current paper is the *first* to address the general issue of why SSA is better than CFG in generic and quantitative terms.

3 Software Metrics

This section describes the various metrics we use, and how we adapt these metric definitions for SSA. Our objective is to use software metrics to measure the differences between CFG and SSA representations of a program. So this means that absolute metric values are not important; we are only concerned with the relative values for CFG and SSA representations of same program.

The main novelty of our approach is that we apply *software metrics* to *compiler IR code* rather than to human-generated, high-level source code. Although previous work has analysed auto-generated code of various kinds [37, 36, 25] this is the first application of software metrics to compiler IR code. We expect this is due to the limited interaction between the software metrics and program analysis research communities.

This section focuses on *intra-procedural* metrics. SSA and CFG are intra-procedural representations. Although inter-procedural versions exist [23, 35] they are not in widespread use.

Since the CFG and SSA representations of a procedure share the same control flow structure, we cannot use control flow metrics like cyclomatic complexity [24] to compare them. Instead, we must concentrate on metrics that depend on *variable naming* conventions. Note that we are considering compiler variables (sometimes known as temporaries or virtual registers), rather than source code variables.

3.1 Size Metric

We measure a procedure's size by the *number of basic blocks* in the graph, i.e. $|B|$ for a CFG $(B, E, b_{\text{entry}}, b_{\text{exit}})$. This size metric is more appropriate for graph-based IRs than the most common metric: source lines of code. Note that a procedure will have the *same size* in both SSA and non-SSA forms.

In SSA program analysis, computational complexity measures are generally size related. For a program of size n, the time complexity of the standard SSA transformation algorithm is $O(n^2)$. The number of SSA variables and number of ϕ-functions are also $O(n^2)$.

3.2 Halstead Metrics

Halstead's complexity metrics [12] were originally developed to measure the complexity of program modules directly by analysis of source code. They are among the earliest software metrics. The Halstead measures are based on four integer values:

1. $n1$—the number of distinct operators

2. $n2$—the number of distinct operands
3. $N1$—the total number of operators
4. $N2$—the total number of operands

In our case, we take *operands* to be virtual registers in the compiler IR code, and *operators* to be instructions and pseudo-instructions. Appendix A gives a full description of how we interpret operators and operands in our metric calculations.

From these four basic values, five complexity metrics are derived, as shown in Table 1.

Table 1. Table of Halstead Complexity Metrics

Measure	Formula
Program length	$N = N1 + N2$
Program vocabulary	$n = n1 + n2$
Volume	$V = N * (\log_2 n)$
Difficulty	$D = (n1/2) * (N2/n2)$
Effort	$E = D * V$

Measurements $n1$ and $N1$ concern number of operators. These numbers will be identical for CFG and SSA programs if we exclude ϕ-functions from consideration.[5] Therefore the only differences occur with $n2$ and $N2$, which concern the names of operands. $n2$ should change from CFG to SSA, since it measures number of distinct operands. On the other hand, $N2$ should remain unchanged, since SSA only renames *existing* operands and does not introduce new instructions. (Again, we exclude ϕ-functions from consideration.) Generally $n2$ will be greater for SSA than CFG, which means that SSA volume V should be larger than CFG, and SSA difficulty D should be smaller than CFG. Since the volume increase is logarithmic, whereas the difficulty decrease is linear, the overall SSA effort E should also decrease in relation to CFG effort.

A high effort score is undesirable; it means that a program module is difficult to understand and maintain. Section 5.1 presents our empirical results for analysis using these Halstead metrics.

3.3 Information Flow Complexity Metric

Henry and Kafura present the information flow complexity metric (IFC) [15] as a *system-level* design metric. It provides a measure of the information that flows between the various modules in a system. In the original study [15] it was applied to source code modules in UNIX. IFC has since been used for system specification

[5] We consider this to be reasonable, since ϕ-functions are only copies rather than real computational operations. A simple copy instruction does not count as an operator, so neither does a ϕ-function.

studies [18] and iterative software design [16]. IFC is often used at a lower level than system modules; it is very commonly used to quantify information flowing between source-level procedures in a program [21]. IFC was deemed to be a measure that could be used to 'produce reliable software' enshrined in IEEE Standard 982.2.

IFC depends on the amount of information flowing into a module, known as fanIn, through parameters and reads of global data structures. IFC also depends on the amount of information flowing out of a module, known as fanOut, through return values and writes to global data structures. Finally IFC depends on the length of the module. The original formula for calculating the IFC for a module is given in equation 1.

$$\text{IFC} = \text{length} * (\text{fanIn} * \text{fanOut})^2 \tag{1}$$

A high IFC score for a module is undesirable. Long modules, and those involved in lots of information flows have high IFC. The message of the original paper is that high IFC indicates *lack of maintainability*. Lots of cross-module dependencies make it difficult to change the system. This is similar to the more recent notion of *coupling* in object-oriented software development [7].

In this study, we adopt IFC to measure the complexity of individual basic blocks, at an intra-procedural level. Although it does not seem that Henry and Kafura's Information Flow metric is immediately applicable here, we re-interpret the SSA representation (rather than modify the metric) to make the application straightforward.

Procedural information flow relies on measuring number of *input* and *output* variables in each procedure. This simply equates to parameters as inputs and return values as outputs in most high-level languages. A single procedure in SSA can be viewed as a collection of smaller functions in a call graph, where each basic block from the original procedure now corresponds to a function in the call graph. Kelsey [20] and Appel[3] give the details of this transformation. They argue that SSA is a form of *functional programming*. Each basic block from the original procedure now becomes a *function*. Upwardly exposed uses (variables used in a basic block that are not previously defined in that same block) become input parameters. Variables whose definitions in this basic block reach to other blocks (variables that live on exit from this basic block) become output parameters. We transform our SSA procedures into these kinds of functional programs, then apply the IFC metric to each function.

Recalling the definition of IFC from equation 1, we now need to clarify how to evaluate it on a given function.

- The length is the number of operations in the function, which should correspond to the instructions in the original basic block.
- The fanIn is the number of input parameters for the function, which should correspond to the number of upwardly exposed variable uses in the original basic block.

– The fanOut is the number of output parameters for the function, which should correspond to the number of variables live on exit from the original basic block.

We aim to compute IFC scores for basic blocks in CFG and SSA IRs, and compare them. To make this comparison, we analyse functions with their SSA variable names to measure fanIn and fanOut, and to compute IFC. Then we remove subscripts from the SSA variable names to recover an approximation to the original CFG variable names. Then we measure fanIn and fanOut for these CFG variables, and compute IFC scores for the functions with CFG variable names. Section 5.2 presents our empirical analysis using the IFC metric. Appendix B gives full details of the IFC calculation.

IFC enables us to measure *variable re-use*. In SSA, variable names are an infinite resource; we instantiate a new variable name at each definition point in the program text. This is not generally the case in standard CFG code. The CFG representation allows variable recycling, since it does not have a strict renaming scheme like SSA. We say that CFG variable *recycling* occurs when several definitions are multiplexed onto the same name. Sometimes this is due to higher level programmer concerns, for example the definitions all relate to the same concept, such as `currentTemperature`. On the other hand, the recycling may be entirely *co-incidental*, for example the programmer (or compiler) re-uses a `tmp` variable as a place-holder for a non-trivial arithmetic expression. However variable re-use can reduce *precision* in data flow analysis, particularly for flow-insensitive data flow analysis. Less variable re-use should make data flow analysis easier, so IFC scores somehow act as a measure of data flow analysis complexity. We want to compare the IFC metrics for programs in SSA and CFG representations.

4 EXPERIMENTAL INFRASTRUCTURE

All intermediate code used in our experimental analysis is generated by the GNU Compiler Collection (GCC) version 4.2.1, running on x86-64 Linux. The SSA form for each procedure is extracted with the `-fdump-tree-ssa` flag. This code is set to be optimized at the `-O3` level, however the SSA dump occurs before most of the optimization takes place. We construct the equivalent CFG code by simply eliding SSA variable subscripts and ϕ-functions.

We create custom perl scripts to analyse the GCC-generated SSA dumps. Our scripts extract the key parts of each SSA dump, namely operators and operands in each basic block, without needing to parse the entire debug dump information. This partial parsing is accomplished using techniques based on *island grammars* [27]. We treat each GCC virtual operand as a variable, in our metrics. We treat each GCC machine operation as an operator, in our metrics. Appendix A gives a full description of the metrics calculations.

The programs to be analysed are the C language programs in the integer section of the SPEC CPU 2000 benchmark suite [14]. Table 2 summarises the

procedural properties for each benchmark. Sizes are measured in terms of number of distinct basic blocks in each procedure. This measure is invariant of the CFG/SSA transformation. Note that the *176.gcc* benchmark is by far the largest in terms of number of procedures to analyse; it also has the largest single procedure.

Table 2. Table of SPEC benchmarks used in experiments, with details of their procedure sizes

benchmark	# *procs*	*proc size*		
		min	*max*	*mean*
164.gzip	67	1	148	27.6
175.vpr	175	0	547	29.9
176.gcc	1807	1	1821	46.1
181.mcf	26	1	106	20.2
186.crafty	39	1	370	60.0
197.parser	323	1	435	27.3
254.gap	698	1	204	29.2
255.vortex	499	1	213	17.5
256.bzip2	74	1	214	17.8
300.twolf	190	1	409	47.3

5 Analysis

This section reports on several metric comparisons between CFG and SSA, using the software metrics described in Section 3 and the tools described in Section 4.

5.1 Halstead Effort Comparisons

This investigation studies the relative difference in Halstead effort between CFG and SSA versions of the each procedure.

The average *ratio* of SSA effort to CFG effort can be computed by dividing the total SSA effort for all 3893 procedures by the total CFG effort for the same procedures. In effect, this is an arithmetic mean, where each procedure's contribution is *weighted* by its original CFG effort. Such a calculation generally rewards reductions for *higher effort* procedures, since these can cause greater efficiency savings. This average SSA:CFG ratio is computed as 0.63, which means that the SSA effort is 37% lower than the CFG effort.

Figure 1 shows how the CFG to SSA Halstead effort ratio changes with program size. Again note the logarithmic scale on the x-axis. Each procedure is denoted by a single (x, y) point, where x is the procedure size and y is the CFG:SSA effort ratio. Points above the line $y = 1$ indicate that the procedure has a lower complexity in SSA than in CFG form. The *majority* of points fall

above $y = 1$. The curve shown on the graph is a linear best-fit computed by linear least-squares regression. This best-fit curve indicates that the magnitude of the complexity reduction for SSA transformation *increases* with procedure size.

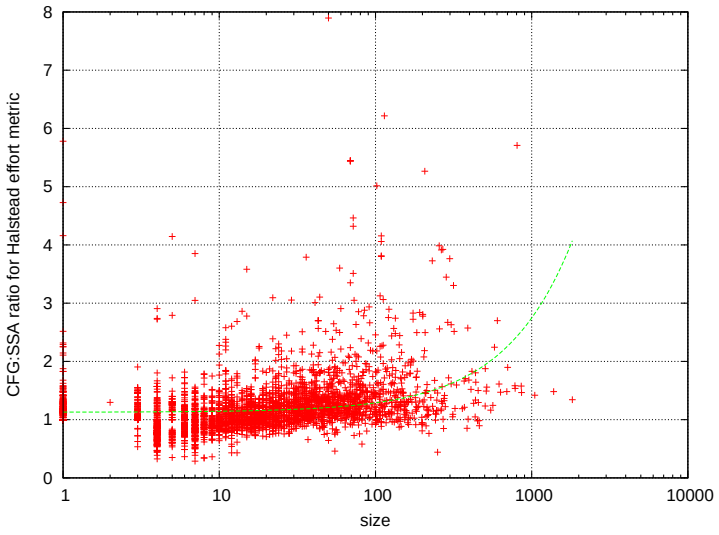


Fig. 1. Correlation of CFG:SSA Halstead effort ratios with procedure size

We report the reduction in total effort for each benchmark program as a result of the SSA transformation. For a single benchmark b, this value is computed by summing the CFG Halstead effort scores for all procedures in b, then summing the SSA Halstead effort scores for all procedures in b, then reporting the relative value of the sum of SSA scores in relation to the sum of CFG scores. Thus a relative value of 1.0 means the SSA transformation does not affect Halstead complexity, for this benchmark. A relative value *below* 1.0 means that the SSA transformation *reduces* the Halstead complexity, for this benchmark. Figure 2 shows these results. The Halstead effort is reduced by the SSA transformation, for all benchmarks. As shown in the graph, the geometric mean of the complexity ratio is 0.68, which means that the average complexity reduction for a SPEC benchmark program in SSA form is 32%, according to Halstead's effort metric.

5.2 Information Flow Complexity Comparison

The final investigation studies the relationship between IFC scores for SSA and CFG versions of each basic block in each procedure.

From our set of benchmarks, there are 3893 procedures. The total number of basic blocks over all these procedures is 141513. We *adjust* the IFC metrics

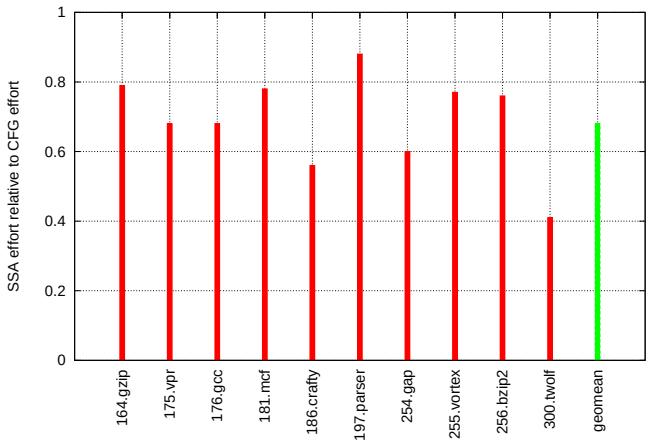


Fig. 2. Reduction in Halstead Effort after the SSA transformation (smaller is better).

scores by adding one to each individual score. This avoids problems with zero values when we compute SSA:CFG ratios and plot log-scale graphs.

The mean SSA:CFG ratio for IFC over all analysed basic blocks is 0.91. Figure 3 shows the results. There is one (x, y) point for each basic block. The x-axis gives the adjusted IFC score for the SSA version of a basic block; the y-axis gives the adjusted IFC score for the corresponding CFG version. Note that 90% of basic blocks have the same IFC score for both IRs. Where there is a difference, then the CFG score is always higher than the SSA score. Thus all points are on or above the line $y = x$. Note the logarithmic scales on both axes in this graph.

We had expected more than 10% of the basic blocks to have lower IFC for SSA than for CFG. The reason for this low proportion is that: (i) many basic blocks are extremely small and simple, so there is no difference between the CFG and SSA forms; and (ii) in the GCC compiler, the CFG creation phase is allowed to use unlimited virtual operands, like SSA creation (although without the possibility of ϕ-functions). As Cooper and Torczon explain [8] this is an attempt to reduce incidental sharing. However, there is still some reuse that can only be eliminated by transformation from CFG to SSA.

There are several interesting trends in the graph in Figure 3. For instance, some basic blocks have an IFC score of zero for SSA; although such scores are adjusted to one for reasons mentioned earlier. This anomaly generally occurs because the SSA fanOut score for these basic blocks is zero, implying that there are no outgoing live variables from the blocks in SSA. Such blocks define only global variables. Due to issues of pointer aliasing and inter-procedural optimization, the GCC variant of SSA gives each use of a global variable a separate subscripted virtual operand name. Thus definitions of global variables (marked with the `VDEF`

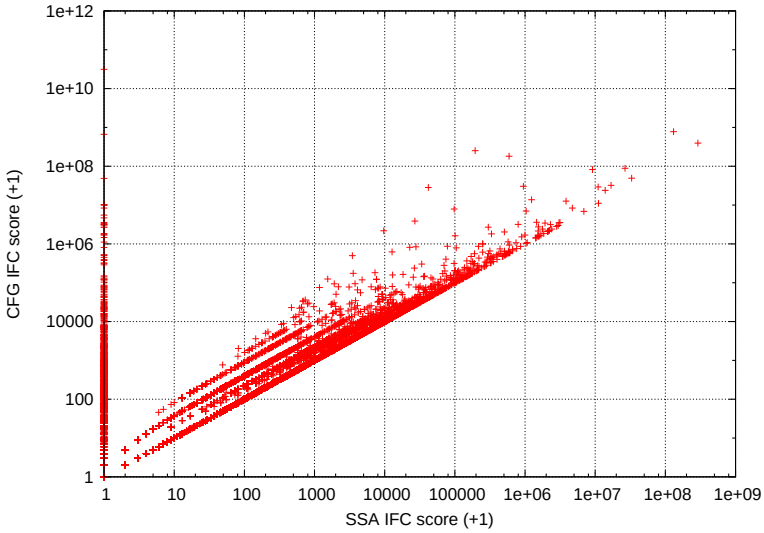


Fig. 3. Comparison of Information Flow Complexity scores for basic blocks in SSA and CFG forms

pseudo-operation in GCC's SSA form) appear to be defining variables that are never used. Thus the fanOut is computed to be zero. It is legitimate to question whether this interpretation of IFC is fair. We argue that the answer is *yes*, since separate subscripts enable data flow disambiguation, which makes it easier to analyse inter-procedural data flows for global variables. In contrast, the CFG representation does not split namespaces for global variables, so inter-procedural analysis is correspondingly less efficient or effective.

For the 10% of basic blocks that have greater IFC for CFG than SSA, in some cases the CFG IFC score is *orders of magnitude* greater, due to the squared term in the IFC equation.

As with Halstead effort above, we also report the reduction in metrics scores for each individual benchmark program as a result of the SSA transformation. For a single benchmark b, this value is computed by summing the CFG IFC scores for all basic blocks in b, then summing the SSA IFC scores for all basic blocks in b, then reporting the relative value of the sum of SSA scores in relation to the sum of CFG scores. Thus a relative value of 1.0 means the SSA transformation does not affect IFC. A relative value *below* 1.0 means that the SSA transformation reduces IFC. Figure 4 shows these results. The IFC metric is reduced by the SSA transformation, for all benchmarks. The geometric mean of the SSA:CFG complexity ratio is 0.4, which means that the average complexity reduction for a SPEC benchmark program in SSA form is 60%, according to the IFC metric.

Note that the complexity reduction is most dramatic for *176.gcc* and *254.gap*. These benchmarks have several data structure initialization procedures with

extremely long basic blocks. The long blocks have high CFG fanOut scores but significantly lower SSA fanOut scores. The squared term in the IFC equation accounts for the massive difference in total complexity.

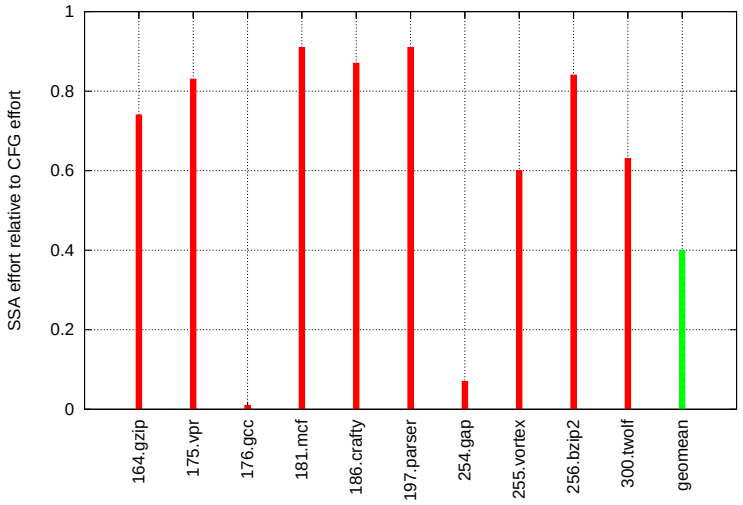


Fig. 4. Reduction in Information Flow Complexity after the SSA transformation (smaller is better).

6 Discussion

The analyses in Section 5 show that the translation of a program from CFG to SSA generally *reduces its complexity* according to two standard metrics.

1. The average per-benchmark Halstead effort metric is 32% lower.
2. The average per-benchmark IFC metric is 60% lower.

Low scores for Halstead and IFC metrics usually indicate well-constructed and easily maintained code. This observation is conventionally applied to high-level source code, where low metrics values indicate that the code should be straightforward for programmers (or software analysis tools, e.g. [22]) to *understand* and *modify*. We argue that the same observation may be applied to low-level intermediate code, where low metrics values indicate that the code should be straightforward for compilers to *analyse* and *optimize*. Therefore, given the results of our metrics-based analyses in Section 5, we conclude that SSA code is easier for compilers to optimize than equivalent CFG code. Intuitively, SSA-based optimization is simpler and more elegant.

The experience of the compiler construction community harmonizes with such a conclusion. They also observe that SSA facilitates better compiler analysis and optimization passes. For instance, Muchnick [28] states:

> SSA ... simplifies and makes more effective several kinds of optimizing transformations, including constant propagation, value numbering, invariant code motion and removal, strength reduction, and partial redundancy elimination.

Again, compiler researchers prefer SSA over CFG because SSA introduces more local variables, which can lead to more precise data flow information being associated with each variable. This is especially true for *flow-insensitive analysis*, where one unit of data flow information is stored for each variable globally; which contrasts with *flow-sensitive* analysis, where there are N_v units of data flow information for each variable v (one for each of the N_v locations where v is in scope). For example, Hasti and Horwitz [13] show that the SSA transformation makes flow-insensitive pointer analysis as accurate as flow-sensitive analysis in some cases. Our metrics-based analysis reinforces this intuition: that SSA is superior to CFG because it has more variables. Halstead's effort metric captures this property, since the effort score is inversely proportional to the number of unique variable names (n_2). The original motivation is that information is assumed to be spread uniformly over all the variable names, which makes it easier analysis to generate precise information about a single name when there are *more variable names*.

Some members of the compiler community raise the objection that SSA adds overhead to the IR, not only due to the expanded vocabulary but also because of the many ϕ-function insertions required. To quote Muchnick [28] again:

> The occasional large increase in program size is of some concern in using SSA form but, given that it makes some optimizations much more effective, it is usually worthwhile.

This is clearly a valid concern; there is some baggage that comes with SSA. However as Muchnick states, the benefit outweighs the cost in general. This tradeoff is obvious in the Halstead metrics. The *volume* metric increases with the CFG to SSA transformation, due to the extra infrastructure. However the *difficulty* metric decreases due to the increased vocabulary. As observed in Section 3.2, difficulty decreases at a faster rate that volume increases; so the *effort* (which is the product of volume and difficulty) also decreases in general.

Another common qualitative justification for SSA is that it localises data flows, due to aggressive *live range splitting* [5, 33]. This reduces accidental sharing of variable names. It means that variable optimizations may have fewer global side-effects. In SSA, data flow optimizations can often occur at the peephole optimization level due to the localization of data flows. Our new interpretation of the IFC metric (at the basic block level) quantifies this localization. Our analysis shows that the CFG to SSA transform does reduce static data flow dependences between basic blocks.

We note one *caveat* from our analyses. As mentioned earlier, the GCC translator from high-level source code to CFG intermediate code (known as the *gimplifier* [26, 31]) is allowed to introduce unlimited new variable names. Although the CFG IR does *not* have the single assignment property before its conversion to SSA, it often has simplified live ranges in relation to the original high-level program. Thus our metrics-based complexity reduction from CFG to SSA should be treated as a *lower bound*. If the gimplifier did not introduce large numbers of new variable names, then the complexity reduction would be even greater for SSA.

However perhaps this would not be a fair evaluation of CFG, since its variable naming convention is often far from naive. For instance, Cooper and Torczon [8] motivate and describe their CFG-based variable naming scheme for a 1980's FORTRAN compiler before they became aware of SSA:

> Unfortunately, associating multiple expressions with a single temporary name obscured the flow of data and degraded the quality of the optimization. The decline in code quality overshadowed any compile-time benefits. Further experimentation led to a short set of [variable renaming] rules that yielded strong optimization while mitigating growth in the name space. ... The compiler used these rules until we adopted SSA form, which has its own naming discipline.

In a similar way, the CFG intermediate code generation process in GCC does some principled variable renaming. Nevertheless, no matter how much variable renaming is applied to CFG; apart from the SSA constraint, there is still potential for unnecessary inefficiency in compiler analyses.

So, if it were needed, this paper provides clear quantitative evidence to endorse the adoption of SSA in GCC.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, 2nd edn. (2006)
2. Amme, W., von Ronne, J., Franz, M.: Quantifying the benefits of ssa-based mobile code. Electronic Notes in Theoretical Computer Science 141(2), 103–119 (2005)
3. Appel, A.W.: SSA is functional programming. ACM SIGPLAN Notices 33(4), 17–20 (Apr 1998)
4. Braun, M., Hack, S.: Register spilling and live-range splitting for SSA-form programs. In: Proceedings of the International Conference on Compiler Construction. Lecture Notes in Computer Science, vol. 5501. Springer (2009)
5. Briggs, P.: Register allocation via graph coloring. Ph.D. thesis, Rice University (1992)
6. Burke, M.G., Choi, J.D., Fink, S., Grove, D., Hind, M., Sarkar, V., Serrano, M.J., Sreedhar, V.C., Srinivasan, H., Whaley, J.: The jalapeño dynamic optimizing compiler for java. In: Proceedings of the ACM 1999 Conference on Java Grande. pp. 129–141 (1999)
7. Chidamber, S., Kemerer, C.: A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20(6), 476–493 (1994)

8. Cooper, K.D., Torczon, L.: Engineering a Compiler. Morgan Kaufmann (2004)
9. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: An efficient method of computing static single assignment form. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 25–35 (1989)
10. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems 13(4), 451–490 (Oct 1991)
11. Fitzgerald, R., Knoblock, T., Ruf, E., Steensgaard, B., Tarditi, D.: Marmot: An optimizing compiler for Java. Software: Practice and Experience 30(3), 199–232 (2000)
12. Halstead, M.H.: Elements of Software Science. Elsevier (1977)
13. Hasti, R., Horwitz, S.: Using static single assignment form to improve flow-insensitive pointer analysis. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 97–105 (1998)
14. Henning, J.L.: SPEC CPU2000: Measuring CPU performance in the new millennium. IEEE Computer 33(7), 28–35 (2000)
15. Henry, S., Kafura, K.: Software structure metrics based on information flow. IEEE Transactions on Software Engineering 7(5), 510–518 (1981)
16. Henry, S., Selig, C.: Predicting source-code complexity at the design stage. IEEE software 7(2), 36–44 (1990)
17. Holloway, G.: The Machine-SUIF static single assignment library (2001), `http://www.eecs.harvard.edu/hube/software/nci/ssa.html`
18. Ince, D., Shepperd, M.: An empirical and theoretical analysis of an information flow-based system design metric. In: Proceedings of the European Conference on Software Engineering. Lecture Notes in Computer Science, vol. 387, pp. 86–99 (1989)
19. Johnson, R., Pingali, K.: Dependence-based program analysis. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 78–89 (1993)
20. Kelsey, R.A.: A correspondence between continuation passing style and static single assignment form. ACM SIGPLAN Notices 30(3), 13–22 (Mar 1995)
21. Laird, L.M., Brennan, M.C.: Software measurement and estimation: a practical approach. Wiley (2006)
22. Lajios, G., Schmedding, D., Volmering, F.: Supporting language conversion by metric based reports. In: Proceedings of the 12th European Conference on Software Maintenance and Reengineering. pp. 314–316 (2008)
23. Liao, S.W., Diwan, A., Bosch, Jr., R.P., Ghuloum, A., Lam, M.S.: SUIF explorer: an interactive and interprocedural parallelizer. In: Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 37–48 (1999)
24. McCabe, T.: A complexity measure. Proceedings of the 2nd International Conference on Software Engineering (1976)
25. McDonald, P., Strickland, D., Wildman, C.: Estimating the effective size of auto-generated code in a large software project. In: Proceedings of the 17th International Forum on COCOMO and Software Cost Modeling (2002)
26. Merrill, J.: GENERIC and GIMPLE: A new tree representation for entire functions. In: Proceedings of the First Annual GCC Developers' Summit. pp. 171–179 (2003), `ftp://gcc.gnu.org/pub/gcc/summit/2003/GENERIC%20and%20GIMPLE.pdf`
27. Moonen, L.: Generating robust parsers using island grammars. In: Proceedings of the Eighth Working Conference on Reverse Engineering. pp. 13–22 (2001)

28. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)
29. Mycroft, A.: Type-based decompilation. In: Proceedings of the 8th European Symposium on Programming. Lecture Notes in Computer Science, vol. 1576, pp. 208–223. Springer (1999)
30. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer (1999)
31. Novillo, D.: Tree SSA—a new optimization infrastructure for GCC. In: Proceedings of the First Annual GCC Developers' Summit. pp. 181–193 (2003), http://www.airs.com/dnovillo/Papers/tree-ssa-gccs03.pdf
32. Novillo, D.: Design and implementation of Tree SSA. In: Proceedings of the Second Annual GCC Developers' Summit. pp. 119–130 (2004), http://www.airs.com/dnovillo/Papers/tree-ssa-gcc2004.pdf
33. Quintão Pereira, F.M., Palsberg, J.: Register allocation by puzzle solving. In: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 216–226 (2008)
34. Schouten, D., Tian, X., Bik, A., Girkar, M.: Inside the Intel compiler. Linux Journal 2003(106), 4 (2003), http://www.linuxjournal.com/article/4885
35. Staiger, S., Vogel, G., Keul, S., Wiebe, E.: Interprocedural Static Single Assignment Form. In: Proceedings of the 14th Working Conference on Reverse Engineering. pp. 1–10 (2007)
36. Succi, G., Liu, E.: A relations-based approach for simplifying metrics extraction. Applied Computing Review 7(3), 27–32 (1999)
37. Ward, M., Bennett, K.: Formal methods to aid the evolution of software. International Journal of Software Engineering and Knowledge Engineering 5(1), 25–47 (1995)
38. Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. ACM Transactions on Programming Languages and Systems 13(2), 181–210 (Apr 1991)

A Operators and Operands in our Metrics Calculations

We have precise definitions of what constitutes an operator or an operand, for the GCC intermediate code that we analyse. This allows us to apply the Halstead metrics consistently to all procedures that we analyse.

Operands are real and virtual statement operands from GCC Tree SSA [32]. A *real* operand represents a single, non-aliased, memory location which is atomically read or modified by a statement (i.e., variables of non-aggregate types whose address is not taken). A *virtual* operand represents either a partial or aliased reference (i.e., structures, unions, pointer dereferences and aliased variables).

Operators are basic GIMPLE three-address operators, as outlined in Table 3. Each operator takes in one or more operand values and performs a single computational operation.

B Calculation of Information Flow Complexity Metric

The IFC metric value for a function (that was originally a basic block in an SSA procedure) depends on three fundamental measures: length, fanIn and fanOut.

Table 3. GIMPLE operators

operator(s)	*description*
[ ]	array subscript
+, −, ∗, /	arithmetic, pointer deref
&, |, ˜, ˆ	logical
<<, >>	shift
==, ! =, <, >, <=, >=	comparison
`case`	switch statement test
`VUSE, VMAYDEF, VMUSTDEF,PHI`	Tree SSA pseudo-operators

The length is computed as the number of GIMPLE operators in the function. Since GIMPLE is a three-address code, there is generally one operator per instruction.

Given a function f that is derived from original basic block b, the fanIn is computed as the number of input parameters for f. There is one input parameter to f for each variable that has an upwardly exposed use in b, whose variable definition dominates b. Again, there is one input parameter to f for each variable that is defined by a ϕ-function at the head of b. (Appel [3] explains how variables defined on the left hand side of ϕ-functions become formal parameters in function definitions, and variables used on the right hand side of ϕ-functions become actual parameters in function calls.)

The fanOut for f is computed as the number of variables defined in f that are used in another function, plus the number of return values for f. Note that some functions define variables that are never used elsewhere. This may be because the live range of such a variable is restricted that single function. Another reason is that GCC has special *virtual operands*. Each aliased memory location has a distinctly named virtual operand for every static occurrence of that location: i.e. each aliased location has an associated virtual operand that is redefined at every potential definition or use of that location. Such virtual operands may appear to be defined and never used. Section 5.2 discusses how we handle such operands.

A New Intermediate Representation for GCC based on the XARK Compiler Framework *

José M. Andión, Manuel Arenaz, and Juan Touriño

Computer Architecture Group
Department of Electronics and Systems
University of A Coruña, A Coruña, Spain
{jandion,arenaz,juan}@udc.es

Abstract. The automatic generation of efficient parallel code continues to be a challenge for compiler technology. In the past, only the developers of applications targeted for supercomputers were aware of exploiting the parallelism available in the hardware. Currently, the increase in the number of cores available in commodity processors has generalized the problem. Parallelizing compilers typically address automatic detection of parallelism through the analysis of a set of standard graphs that capture information about the statements of the program. Well-known examples are the Data Dependence Graph (DDG), the Control Flow Graph (CFG) and the Dominance Tree (DT). This paper outlines a new compiler intermediate representation (IR) intended to ease detection of parallelism in sequential programs. In constrast to standard compiler IRs that are based on the concept of program statement, the new IR hinges on the concept of *computational kernel* provided by the XARK compiler framework. Thus, a kernel represents a program construct that is used frequently by programmers. Well-known examples are inductions, reductions or recurrences. In addition, this paper presents several ideas for using such kernel-level IR for the automatic generation of parallel code for multi-core processors. An application from the SPEC CPU2000 benchmark suite is used as case study.

1 Introduction

Even for experienced programmers, the development and maintenance of programs that make an efficient use of the computer architecture is a complex time-consuming task. The actual emergence and widespread use of multi-core processors adds more complexity, as current compiler technology cannot generate efficient parallel code for multi-core processors, being the programmer responsible for exploiting the parallelism available in hardware.

Current optimizing and parallelizing compilers split programs into statements that are represented as abstract syntax trees. In order to capture program behavior, the compiler builds an intermediate representation that consists of a

* This research was supported by the Ministry of Education and Science of Spain and FEDER funds of the European Union (Project TIN2007-67537-C03) and by the Galician Government (Project PGIDIT06PXIB105228PR)

set of standard graphs that connect program statements [1, 2, 10]. For instance, the GCC compiler establishes relationships between GIMPLE-SSA statements by building a control flow graph (CFG), a data dependence graph (DDG) or a dominance tree (DT). Although this approach was shown to be effective in practice for compilation of real codes for mono-core processors, the level of abstraction of statement-based IRs makes it difficult for the compiler to detect the parallelism available in real applications.

The recognition of program constructs that are frequently used by software developers (*computational kernels*, from now on) is a powerful mechanism to detect parallelism automatically [4, 6–9]. XARK [5] is an extensible compiler framework that provides a complete, robust and general solution to the problem of automatic recognition of computational kernels (inductions, reductions or array recurrences), even in the presence of complex control flows.

This paper presents preliminary work on the proposal of a new kernel-based IR and sketches an algorithm to guide automatic parallelization for multi-core processors. As an illustrative case study, the full-scale application EQUAKE of the SPEC CPU2000 benchmark suite is analyzed.

The rest of the paper is organized as follows. Section 2 describes the EQUAKE application. Section 3 presents the XARK compiler framework and the collection of kernels recognized in EQUAKE. Section 4 sketches a new kernel-based IR and describes the way it captures the behavior of EQUAKE as a whole. Section 5 outlines a preliminary algorithm for the automatic parallelization of EQUAKE targeting multi-core processors. Finally, Section 6 concludes the paper and presents future work.

2 The Application EQUAKE of SPEC CPU2000

One of the applications of the SPEC CPU2000 benchmark suite is *EQUAKE*. This program performs a simulation of seismic waves in large, highly heterogeneous valleys. EQUAKE is capable of recovering the time history of the ground motion due to a specific seismic event everywhere within a valley. Computations are carried out on an unstructured mesh which locally resolves wavelengths using a finite element method. The source code of EQUAKE consists of 1512 lines grouped into 16 procedures. This relatively-small size allows us to analyze its behavior manually in a reasonable amount of time. Our analysis of EQUAKE revealed that the code contains a significant variety of the kernels that are used frequently in practice.

For the sake of clarity, Figures 1 and 2 show only an excerpt of the `main()` and `smvp()` procedures, respectively. Note that `smvp()` is the most costly routine of the program. Approximately, it consumes more than 70% of the total execution time and, therefore, it is a likely target for an optimizing compiler. The loop `for`$_i$ of `main()` procedure (see Figure 1, lines 11–26) traverses the set of finite elements in order to assemble the global simulation variables `M` and `C` using the contribution of each finite element to the solution. In each `for`$_i$ iteration,

the individual contribution is computed and stored in the element matrices, for instance, matrix `Ce`.

The output of EQUAKE is a report of the displacements at both the hypocenter and epicenter of the earthquake for a predetermined number of simulation timesteps. For this purpose, the loop nest $\texttt{for}_{iter}$ (Figure 1, lines 29–59) performs a time integration. In each timestep, the displacement (3D array `disp`) is computed using the values corresponding to the two previous timesteps and involving several procedure calls at run-time (e.g., `smvp()`). The arrays computed during the simulation phase (e.g., `M` and `C`) are used as input data. Once the displacement `disp` has been computed, each $\texttt{for}_{iter}$ iteration finishes by calculating the velocity (2D array `vel`). The three variables `disptminus`, `dispt` and `disptplus` are updated in a round-robin fashion (i.e., a circular swap) in each iteration of $\texttt{for}_{iter}$ (see lines 54–58). These three variables are used to access to the 3D array `disp`, which stores the results of EQUAKE in the current and the previous two timesteps.

3 The XARK Compiler Framework

The XARK compiler framework [5] builds a hierarchical representation of a program through the recognition of a comprehensive collection of kernels. It decomposes the program into a set of mutually dependent kernels that capture the behavior of a code fragment and provide the compiler with information about the computations performed at runtime on scalar and non-scalar variables (e.g., arrays, pointers). A detailed description of the collection of kernels can be consulted in [5]. Next, we present the kernels that appear in EQUAKE:

- *Regular assignment*: Assignment of a new value to a set of elements of an array variable following a regular access pattern, typically given by an affine expression of a loop index. The computation of array `vel` in the time integration phase (Figure 1, lines 49–52), assigns a new value to each element `vel[i][j]`, being `i` and `j` affine expressions of the indices of the enclosing loops $\texttt{for}_i$ and $\texttt{for}_j$. Another example is the computation of array `Ce` in each $\texttt{for}_i$ iteration during the simulation phase (Figure 1, lines 12–14).
- *Regular reduction*: The elements of an array variable are assigned a new value that depends on the value already stored in the array element. The elements updated in a regular reduction are determined by a regular access pattern. The computation of array `Ce` in lines 16–17 of Figure 1 is a regular reduction with a sum (+) reduction operation and a linear access pattern. Another example is the computation of array `disp[disptplus][*][*]` in the loop nest $\texttt{for}_i$ of lines 39–44 of Figure 1. In this case, the new value `disp[disptplus][i][j]` depends on itself and on two different array elements `disp[dispt][i][j]` and `disp[disptminus][i][j]`. Note that in the scope of $\texttt{for}_i$, the variables `disp[disptplus][*][*]`, `disp[dispt][*][*]` and `disp[disptminus][*][*]` represent three 2D arrays that do not overlap in memory.

```
double **M, **C, **M23, **C23, **V23, **vel, ***disp, ***K;
int i, j, k, ii, jj, kk, iter, timesteps, disptplus, dispt, disptminus;
double time, Ke[12][12], Me[12], Ce[12], alpha;

/* Initialization phase */
disptplus = 0;
dispt = 1;
disptminus = 2;

/* Simulation phase */
for (i = 0; i < ARCHelems; i++) {
   for (j = 0; j < 12; j++) {
      Me[j] = 0.0;
      Ce[j] = 0.0;
   }
   for (j = 0; j < 12; j++)
      Ce[j] = Ce[j] + alpha * Me[j];
   for (j = 0; j < 4; j++) {
      M[ARCHvertex[i][j]][0] += Me[j * 3];
      M[ARCHvertex[i][j]][1] += Me[j * 3 + 1];
      M[ARCHvertex[i][j]][2] += Me[j * 3 + 2];
      C[ARCHvertex[i][j]][0] += Ce[j * 3];
      C[ARCHvertex[i][j]][1] += Ce[j * 3 + 1];
      C[ARCHvertex[i][j]][2] += Ce[j * 3 + 2];
   }
}

/* Time integration phase */
for (iter = 1; iter <= timesteps; iter++) {
   for (i = 0; i < ARCHnodes; i++)
      for (j = 0; j < 3; j++)
         disp[disptplus][i][j] = 0.0;
   smvp(ARCHnodes, K, ARCHmatrixcol, ARCHmatrixindex, disp[dispt],
      disp[disptplus]);
   time = iter * Exc.dt;
   for (i = 0; i < ARCHnodes; i++)
      for (j = 0; j < 3; j++)
         disp[disptplus][i][j] *= - Exc.dt * Exc.dt;
   for (i = 0; i < ARCHnodes; i++)
      for (j = 0; j < 3; j++)
         disp[disptplus][i][j] += 2.0 * M[i][j] * disp[dispt][i][j] -
               (M[i][j] - Exc.dt / 2.0 * C[i][j]) * disp[disptminus][i][j] -
               Exc.dt * Exc.dt * (M23[i][j] * phi2(time) / 2.0 +
               C23[i][j] * phi1(time) / 2.0 + V23[i][j] * phi0(time) / 2.0);
   for (i = 0; i < ARCHnodes; i++)
      for (j = 0; j < 3; j++)
         disp[disptplus][i][j] = disp[disptplus][i][j] /
               / (M[i][j] + Exc.dt / 2.0 * C[i][j]);
   for (i = 0; i < ARCHnodes; i++)
      for (j = 0; j < 3; j++)
         vel[i][j] = 0.5 / Exc.dt * (disp[disptplus][i][j] -
                  disp[disptminus][i][j]);

   /* Circular swap */
   i = disptminus;
   disptminus = dispt;
   dispt = disptplus;
   disptplus = i;
}
```

Fig. 1. Excerpt of the source code of the EQUAKE application.

```
1  void smvp(int nodes, double ***A, int *Acol, int *Aindex, double **v,
2            double **w) {
3     int i;
4     int Anext, Alast, col;
5     double sum0, sum1, sum2;
6
7     for (i = 0; i < nodes; i++) {
8        Anext = Aindex[i];
9        Alast = Aindex[i + 1];
10       sum0 = A[Anext][0][0]*v[i][0] + A[Anext][0][1]*v[i][1] +
11              + A[Anext][0][2]*v[i][2];
12       sum1 = A[Anext][1][0]*v[i][0] + A[Anext][1][1]*v[i][1] +
13              + A[Anext][1][2]*v[i][2];
14       sum2 = A[Anext][2][0]*v[i][0] + A[Anext][2][1]*v[i][1] +
15              + A[Anext][2][2]*v[i][2];
16       Anext++;
17       while (Anext < Alast) {
18          col = Acol[Anext];
19          sum0 += A[Anext][0][0]*v[col][0] + A[Anext][0][1]*v[col][1] +
20                  + A[Anext][0][2]*v[col][2];
21          sum1 += A[Anext][1][0]*v[col][0] + A[Anext][1][1]*v[col][1] +
22                  + A[Anext][1][2]*v[col][2];
23          sum2 += A[Anext][2][0]*v[col][0] + A[Anext][2][1]*v[col][1] +
24                  + A[Anext][2][2]*v[col][2];
25          w[col][0] += A[Anext][0][0]*v[i][0] + A[Anext][1][0]*v[i][1] +
26                     + A[Anext][2][0]*v[i][2];
27          w[col][1] += A[Anext][0][1]*v[i][0] + A[Anext][1][1]*v[i][1] +
28                     + A[Anext][2][1]*v[i][2];
29          w[col][2] += A[Anext][0][2]*v[i][0] + A[Anext][1][2]*v[i][1] +
30                     + A[Anext][2][2]*v[i][2];
31          Anext++;
32       }
33       w[i][0] += sum0;
34       w[i][1] += sum1;
35       w[i][2] += sum2;
36    }
37 }
```

Fig. 2. Source code of the `smvp()` procedure of the EQUAKE application.

– *Irregular reduction*: A reduction where the updated array elements are determined by an indirection array. An example is the computation of `M` in the simulation phase (Figure 1, lines 18–21), which uses an indirection array `ARCHvertex` to determine the element of `M` updated in each iteration of $\texttt{for}_i$ and $\texttt{for}_j$.

The automatic recognition of computational kernels in full-scale real application requires the development of an inter-procedural kernel recognition engine. For this purpose, work in progress [3] focuses on the development of an efficient inter-procedural Gated Single Assignment (GSA) form on top of the GIMPLE-SSA infrastructure provided by GCC. GSA is an extension of the well-known Static Single Assignment (SSA) form where reaching definition information of scalar and array variables is represented syntactically. The construction of GSA involves two main tasks: first, placement of special operators (called ϕ generically) at the points of the program with multiple predecessors in the control flow graph; and second, renaming of program variables so that the left-hand sides of the assignment statements define distinct unique variables. Different

kinds of ϕ operators are distinguished according to the point of the program where they are inserted:

- $\mu(x_{out}, x_{in})$, which appears at loop headers and selects the initial x_{out} and loop-carried x_{in} values of a variable.
- $\gamma(c, x_{true}, x_{false})$, which is located at the confluence node associated with a branch (e.g., if-endif construct), and captures the condition c for each definition to reach the confluence node: x_{true} if c is fulfilled; x_{false}, if c is not satisfied.
- $\alpha(a_{prev}, s, e)$, whose meaning is that the element s of an array variable a is set to the value e and the other elements take the values of the previous definition of the array, denoted as a_{prev}.

As shown in Figure 2, `smvp()` carries out an irregular reduction and stores the result in the reduction array `w`, being `Acol` the indirection array and assuming that formal parameters `A`, `v` and `w` point to non-overlapping memory locations. Note that the inter-procedural GSA form enables the recognition of kernels through procedure calls. At the call site of lines 33–34 of Figure 1, the actual parameters of `smvp()` are known to point to disjoint memory locations. Thus, the recognition engine infers that `smvp()` computes an irregular reduction on array `disp[disptplus]` within the main program.

4 A New Kernel-Based Intermediate Representation

In this section we propose a new kernel-based IR intended to expose coarse-grain and fine-grain parallelism to the compiler. Standard IRs typically consist of statement-based DDG, CFG and DT. In a similar manner, our new kernel-based IR consists of a Kernel-based DDG (K-DDG) and a Kernel-based CFG (K-CFG).

As mentioned in Section 3, XARK builds a hierarchical representation that decomposes a program into a set of mutually dependent kernels. We call this representation the K-DDG, whose nodes represent kernels and whose edges represent dependences between kernels. For illustrative purposes consider the graph of EQUAKE presented in Figure 3. The nodes are depicted as ovals labeled with the program variable that stores the results of the computation of the kernel, as well as the type of kernel. On the one hand, the nodes `Me` and `M` of the simulation phase capture two kernels: regular assignment and irregular reduction, respectively. Note that kernels abstract the computations of a set of statements spread over the program. For instance, kernel `M` contains three statements of the body of loop `for`$_j$ (see Figure 1, lines 18–21). On the other hand, the edges capture the kernel-level dependences that connect statements of different kernels (see edge between kernels `Me` and `M` in Figure 3). Note that the remaining dependences between statements of a kernel are not exposed to the compiler in the K-DDG as they are represented in the type of kernel recognized by XARK.

The second graph of our new IR is the Kernel-based Control Flow Graph (K-CFG). We propose a two-phase construction algorithm that first groups the

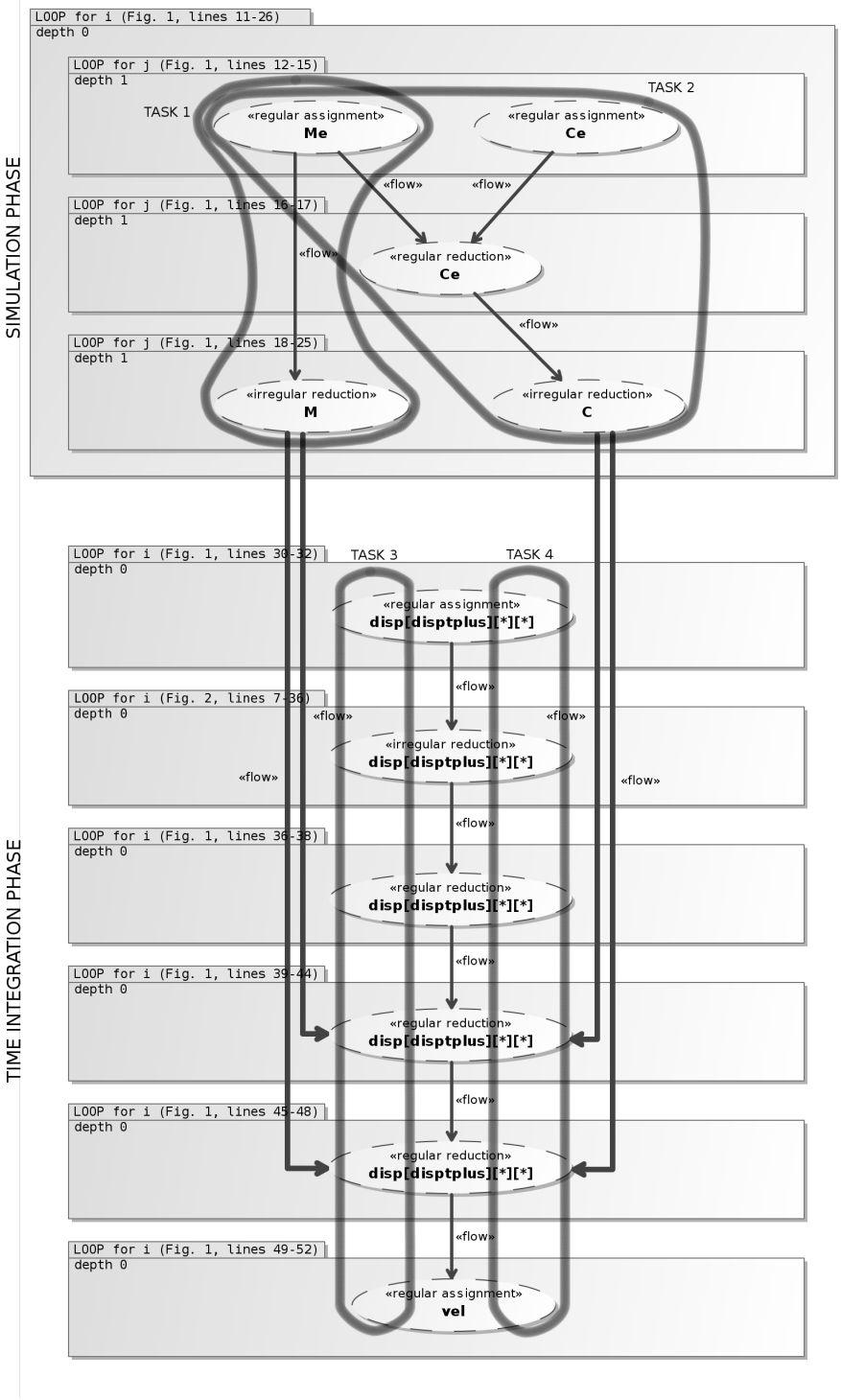


Fig. 3. Kernel-based IR of the source code of the EQUAKE application.

kernels (i.e., the nodes of the K-DDG) provided by XARK into *execution scopes* and later searches for *flow dependences* between the kernels associated with each execution scope. In the first phase, the execution scope of a kernel is computed using the concept of *region* [1]. A region of a flow graph is a collection of nodes N and edges E such that: (1) There is a header h in N that dominates all the nodes in N; (2) If some node m can reach a node n in N without going through h, then m is also in N; and (3) E is the set of all the control flow edges between nodes n_1 and n_2 in N, except (possibly) for some that enter h. Note that loop nests are often the most costly part of a program, thus being the most appropriate candidates for a compiler to extract parallelism. In order to build the K-CFG, the program is split into a hierarchy of loop regions that represent the execution scopes and kernels are attached to execution scopes by executing Algorithm 1. The inputs are the K-DDG provided by XARK and the CFG and the DT provided by the GCC compiler. The algorithm determines the set of basic blocks that contain each statement of a kernel K, excluding μ-statements associated to loop headers. Next, the basic block *bb_dom* that dominates the remaining basic blocks in the set is computed. The γ-statements of the GSA form assure that such basic block *bb_dom* exists. Finally, the execution scope of the kernel K is the innermost loop region that contains *bb_dom*.

In the second phase, the kernel-level dependences of the K-DDG are classified into *flow* or *not flow* dependences using Algorithm 2. For a kernel-level dependence $K_1 \rightarrow K_2$ to be a flow dependence, a dominate relationship between K_1 and K_2 must exist. Two cases are distinguished. In the first case, the execution scopes R_1 and R_2 of K_1 and K_2 are checked. If R_1 and R_2 have the same parent region and R_1 precedes R_2 in the hierarchy (see Algorithm 2, lines 4–5), then all the statements of K_1 are executed before any statement of K_2. Thus, $K_1 \rightarrow K_2$ is a flow dependence that assures that the results of K_1 have been computed before beginning the execution of K_2.

In the second case, we assume that K_1 and K_2 are attached to the same execution scope ($R_1 = R_2$), or that R_1 and R_2 do not have the same parent region. In order to establish a flow dependence between K_1 and K_2, we need to assure that all the statements of K_1 dominate all the statements of K_2 (see Algorithm 2, lines 6–10). An example of this situation is shown below:

```
1 | if (c == 0) {
2 |     a = 5;
3 |     b = a + 3;
4 | } else {
5 |     a = 2;
6 |     b = a + 1;
7 | }
```

In this case, in each path of this if-then statement there is a pair-wise dominance relation between the sentence that writes `a` and the sentence that writes `b`. Thus, Algorithm 2 concludes that the kernel associated to `a` dominates the kernel associated to `b`.

The resulting kernel-based IR for the EQUAKE program is shown in Figure 3. In the K-CFG the nodes are grouped into execution scopes that represent the loops of the code. For the sake of clarity, the execution scopes of the inner loops

Algorithm 1 Computation of the execution scopes.

Input: K-DDG, CFG, DT

```
1: foreach kernel K in the K-DDG do
2:     bb_dom = basic block of CFG that contains a stmt of K (excluding μ-stmt)
3:     foreach statement stmt in K do
4:         if stmt is not a μ-statement then
5:             bb_stmt = basic block of CFG that contains stmt
6:             if bb_stmt dominates bb_dom then
7:                 bb_dom = bb_stmt
8:             end if
9:         end if
10:    end for
11:    K.execution_scope = innermost enclosing loop region of bb_dom;
12: end for
```

Algorithm 2 Detection of kernel-level flow dependences.

Input: K-DDG, K-CFG, CFG, DT

```
1: foreach kernel-level dependence K1 → K2 of the K-DDG do
2:     R1 = execution_scope(K1)
3:     R2 = execution_scope(K2)
4:     if (R1.parent_reg = R2.parent_reg) & (R1 precedes R2 in the hierarchy) then
5:         mark K1 → K2 as flow dependence
6:     else if ∀s1 ∈ K1 ∃s2 ∈ K2 such that statements s1 and s2
7:             belong to the same basic block in the CFG
8:             and s1 precedes s2 in the DT then
9:         mark K1 → K2 as flow dependence
10:    end if
11: end for
```

of the time integration phase are not depicted in Fig. 3 (e.g., execution scope of inner loop $\texttt{for}_j$ — lines 40–44 — in loop nest $\texttt{for}_i$ — lines 39–44 —). As will be shown in the following section, this high-level representation abstracts the implementation details of the computational kernels, enabling the compiler to automatically detect the parallelism available in EQUAKE and to generate parallel code for a multi-core processor.

5 Automatic Detection of Parallelism for Multi-Core Processors

In general, the automatic parallelization of sequential codes involves two main problems: first, detection and decomposition of a program into a set of tasks that can be run in parallel; and, second, generation of parallel code for the underlying hardware architecture. The kernel-based IR presented in Section 4 exposes multiple levels of parallelism that range from parallelizable individual kernels (*intra-kernel parallelism*) up to a kernel-level dependence graph bounded to execution scopes (*inter-kernel parallelism*). Next, we outline a simple algorithm to

Algorithm 3 Task decomposition for multi-core processors.

Input: K-DDG, K-CFG

```
 1: merge execution scopes with one kernel and one cross-boundary edge
 2: d = 0
 3: foreach execution scope R at depth d in the K-CFG do
 4:     if ∀ kernel K ∈ R such that K is parallelizable then
 5:         n_drain_kernels = number of kernels without outgoing edges in K-DDG
 6:                           that cross the execution scope boundaries
 7:         if n_drain_kernels = P then
 8:             tasks = set of P drain kernels
 9:         else if n_drain_kernels < P then
10:             tasks = split parallelizable drain kernels to create P tasks
11:         else
12:             tasks = merge drain kernels to create P tasks
13:         end if
14:         map tasks to different cores
15:     end if
16:     d++
17: end for
```

decompose a program into tasks using current (dual-, quad-, eight-core) multi-core processors.

Algorithm 3 presents a pseudocode of the task decomposition strategy. The analysis focuses on those coarse-grain execution scopes where all the kernels are parallelizable, i.e, there exists a parallelizing code transformation targeted to each kernel [4, 6–9]. The types of kernels found in the K-CFG of EQUAKE are regular assignment, regular reduction and irregular reduction. Thus, all of these execution scopes are analyzed by the task decomposition algorithm.

The simulation phase contains two kernels that lead to the creation of two tasks: the first task corresponds to the execution of the kernels `Me` and `M` (see subgraph TASK1 in Figure 3); the second task involves kernels `Me`, `Ce` and `C` (see subgraph TASK2). If the target multi-core processor contains more than two cores, then intra-kernel parallelism is exploited by applying parallelizing transformations to the irregular reductions `M` and/or `C` as needed (see lines 9–10 in Algorithm 3). Note that, in general, this strategy replicates computations on different cores (see kernel `Me` belonging to TASK1 and TASK2 in Figure 3).

The time integration phase cannot be executed before the simulation phase finishes in order to prevent the violation of cross-boundary kernel-level dependences (appropriate synchronization is needed). The time integration phase is represented by six execution scopes, each containing one kernel. All of these six kernels are connected by one kernel-level flow dependence. As a result, these execution scopes are merged in order to expose a sequence of kernels to the compiler (see line 1 in Algorithm 3). Next, the kernel `vel` that lacks outgoing edges is transformed into parallel code in order to create as many tasks as needed. For illustrative purposes, assume that two tasks TASK3 and TASK4 are created (see Figure 3). Each task computes a subarray of `vel`. Thus, in order to minimize

communication and synchronization, they must be assigned the computation of the corresponding subarrays of `disp`. As a result, the tasks work in parallel with memory locations that do not overlap. Finally, note that kernels with irregular access patterns need to be transformed using an inspector-executor approach to avoid communication and synchronization between the cores.

Overall, the strategy outlined in this section enables the detection of parallelism within full-scale applications. The kernel-based IR (K-DDG and K-CFG) naturally reflects the structure of the source code and, thus, avoids the violation of the data dependences specified by the programmer.

6 Conclusions and Future Work

This paper is a first step towards the definition of a kernel-based IR that exposes multiple levels of parallelism to the compiler. The new kernel-based IR is inspired by standard statement-based IRs used in current optimizing compilers. Thus, the K-DDG and the K-CFG are intended to provide a powerful framework for the development of new full-scale automatic parallelization techniques. The EQUAKE program of SPEC CPU2000 was used as case study to show the potential of this approach.

As future work we intend to improve the K-CFG construction algorithm and to run tests with well-known benchmark suites (e.g., SPEC, PERFECT). In addition, we will address the development of an algorithm for task decomposition that targets multi-core and many-core processors as well as GPU processors.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley (2006)
2. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann (2001)
3. Arenaz, M., Amoedo, P., Touriño, J.: Efficiently Building the Gated Single Assignment Form in Codes with Pointers in Modern Optimizing Compilers. In: 14th International Euro-Par Conference (Euro-Par), Las Palmas de Gran Canaria, Spain. LNCS, vol. 5168, pp. 360–369. Springer (2008)
4. Arenaz, M., Touriño, J., Doallo, R.: Compiler Support for Parallel Code Generation through Kernel Recognition. In: 18th International Parallel and Distributed Processing Symposium (IPDPS), Santa Fe, NM, USA. IEEE Computer Society (2004)
5. Arenaz, M., Touriño, J., Doallo, R.: XARK: An eXtensible framework for Automatic Recognition of computational Kernels. ACM Trans. Program. Lang. Syst. 30(6) (2008)
6. Callahan, D.: Recognizing and Parallelizing Bounded Recurrences. In: 4th International Workshop on Languages and Compilers for Parallel Computing (LCPC), Santa Clara, CA, USA. LNCS, vol. 589, pp. 169–185. Springer (1991)
7. Lin, Y., Padua, D.A.: On the Automatic Parallelization of Sparse and Irregular Fortran Programs. In: 4th International Workshop on Languages, Compilers, and

Run-Time Systems for Scalable Computers (LCR), Pittsburgh, PA, USA. LNCS, vol. 1511, pp. 41–56. Springer (1998)
8. Pinter, S.S., Pinter, R.Y.: Program Optimization and Parallelization Using Idioms. ACM Trans. Program. Lang. Syst. 16(3), 305–327 (1994)
9. Setoain, J., Tenllado, C., Gómez, J.I., Arenaz, M., Prieto, M., Touriño, J.: Towards Automatic Code Generation for GPU Architectures. In: 9th International Workshop on State-of-the-Art in Scientific Computing on GPUs (PARA), Trondheim, Norway (2008)
10. Wolfe, M.: High performance compilers for parallel computing. Addison-Wesley (1996)

Transforming GCC into a research-friendly environment: plugins for optimization tuning and reordering, function cloning and program instrumentation

Yuanjie Huang[1,2], Liang Peng[1,2], Chengyong Wu[1]
Yuriy Kashnikov[4], Jörn Rennecke[3], Grigori Fursin[3]

[1] Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[2] Graduate School of the Chinese Academy of Sciences, Beijing, China
[3] INRIA Saclay, Orsay, France (HiPEAC member)
[4] University of Versailles at Saint-Quentin-en-Yvelines, France

Abstract. Computer scientists are always eager to have a powerful, robust and stable compiler infrastructure. However, until recently, researchers had to either use available and often unstable research compilers, create new ones from scratch, try to hack open-source non-research compilers or use source to source tools. It often requires duplication of a large amount of functionality available in current production compilers while making questionable the practicality of the obtained research results. The Interactive Compilation Interface (ICI) has been introduced to avoid such time-consuming replication and transform popular, production compilers such as GCC into research toolsets by providing an ability to access, modify and extend GCC's internal functionality through a compiler-dependent hook and clear compiler-independent API with external portable plugins without interrupting the natural evolution of a compiler.

In this paper, we describe our recent extensions to GCC and ICI with the preliminary experimental data to support selection and reordering of optimization passes with a dependency grammar, control of individual transformations and their parameters, generic function cloning and program instrumentation. We are synchronizing these developments implemented during Google Summer of Code'09 program with the mainline GCC 4.5 and its native low-level plugin system. These extensions are intended to enable and popularize the use of GCC for realistic research on empirical iterative feedback-directed compilation, statistical collective optimization, run-time adaptation and development of intelligent self-tuning computing systems among other important topics. Such research infrastructure should help researchers prototype and validate their ideas quickly in realistic, production environments while keeping portability of their research plugins across different releases of a compiler. Moreover, it should also allow to move successful ideas back to GCC much faster thus helping to improve, modularize and clean it up. Furthermore, we are porting GCC with ICI extensions for performance/power autotuning for data centers and cloud computing systems with heterogeneous architectures or for continuous whole system optimization.

1 Introduction and Related Work

The compiler is an essential part of modern computing systems responsible for delivering best performing executables across a wide range of architectures quickly and automatically while often satisfying multiple constraints such as code size and compilation time.

Tuning default optimization heuristics of a compiler or optimizing a given program for a given architecture is a tedious, repetitive, error prone, and time-consuming process. In the past few decades, multiple techniques have been developed to improve, automate and speed up this process including empirical iterative feedback-directed compilation [1–11], genetic algorithms and machine learning techniques [12–21], continuous optimization and run-time adaptation [22–28], statistical collective optimization [29, 30] and many other popular methods.

In-house research compilers have been utilized in research for a long time but it is often difficult or even impossible to reproduce their results in realistic environments. Source to source transformation tools such as SUIF [31] and ROSE [32] are also popular to prototype research ideas, however the former is now heavily outdated while the latter is still rapidly evolving, not yet stable enough and is missing some important functionality. We find such frameworks useful for high-level source code manipulation, but we also found that they often have complex interference with the internal optimization heuristics of the coupled source-to-binary compiler making it difficult to analyze final experimental results.

Production proprietary compilers are also regularly used for research. However, they have not been designed to enable prototyping of research ideas, and it is not always easy or possible to access internals of such compilers. Moreover, it is also often impossible to reproduce experimental results in academia without a license. In such cases, researchers may only have access to global compiler flags or some pragmas to tune applications, which may not always be suitable for advanced experimentation.

The LLVM [33] compiler infrastructure also appeared recently targeting both industry and academia, and providing a clear documented API, extension capabilities, JIT, VM, etc. It is gaining popularity but it may still take a long time to provide all available optimizations and support multiple architectures.

GCC [34] is an open-source production compiler that has also been used in research for a while. However, its complexity, often undocumented internals and functions changing from one version to another, long learning curve, rapid evolution, overheads due to frequent synchronization with the mainline compiler and lack of easy extensibility have sometimes prevented it from being used in long-term research projects. Nevertheless, its advantages are very mature and stable multiple front-ends, support for more than 30 families of architectures, GPL license and wide-spread popularity. Moreover, the recently added modular optimization pass manager, experimental polyhedral optimizations (GRAPHITE) and some elementary support for dynamic compilation using CIL [35] and MONO [36] make GCC very attractive for realistic research on code and architecture design and optimizations.

In order to remove some of the above listed disadvantages of production compilers as a research infrastructure and make research developments more portable, we started developing the Interactive Compilation Interface (ICI) [37] to gradually open up compilers and provide access to their internal functionality such as program analysis and optimizations necessary for multiple research projects through a common API and external plugins. It allows quick prototyping of research ideas in a real production environment, potentially saving the effort to build new compiler infrastructure from scratch, while keeping plugin compatibility needed for long-term research projects during natural compiler evolution. GCC maintainers may have some overhead to support such a plugin system, however the GCC community can also benefit from successful research ideas that can be moved back to the compiler immediately. Moreover, it may eventually help to gradually clean up, modularize and document the previously rigid compiler.

ICI has had several major evolutions since 2005 and has been used recently in the long-term MILEPOST project (2006-2009) [19] to add feature extraction passes and enable selection of global optimizations based on popular machine learning techniques. At the beginning it was a compiler-dependent monolithic plugin system, however recently we decided to separate it into 2 parts: low-level compiler dependent plugin system and high-level compiler independent ICI made as a library. The key idea is to update/modify low-level ICI plugin system for different releases of a compiler while keeping high-level ICI reasonably stable to ensure portability of research plugins. In this article, we present further extensions to ICI made during the Google Summer of Code program (GSoC'09) to provide generic function cloning, program instrumentation, pass reordering and control of individual optimizations and their parameters. They are intended to help continue research on various topics including empirical transparent collective optimization [29, 30], run-time program adaptation [25, 21] and code instrumentation, parallelization and scheduling for many-core systems [38, 39, 21].

In the last few years other plugin systems have been proposed and implemented in GCC to enable program analysis, add new passes and control compilation flow [40–42]. Finally, the common agreement has been reached and GCC 4.5 will feature the first common compiler-dependent plugin system. In such case, we can simply substitute low-level compiler-dependent ICI with the native plugin system while keeping high-level ICI compiler-independent that is very important to researchers. We are currently synchronizing our low-level ICI with the plugin system of GCC 4.5 to avoid further duplicate parallel developments. Furthermore, if high-level ICI plugins become stable, they can be easily moved inside the compiler with minimal changes.

The rest of the paper is organized as following: the next section introduces our vision of GCC plugin-enabled research framework, followed by the description of GSoC'09 extensions and some preliminary experimental results. Finally, we briefly describe our attempt to synchronize ICI with the mainline GCC 4.5, followed by a section of conclusions and future work.

2 GCC and collaborative research framework

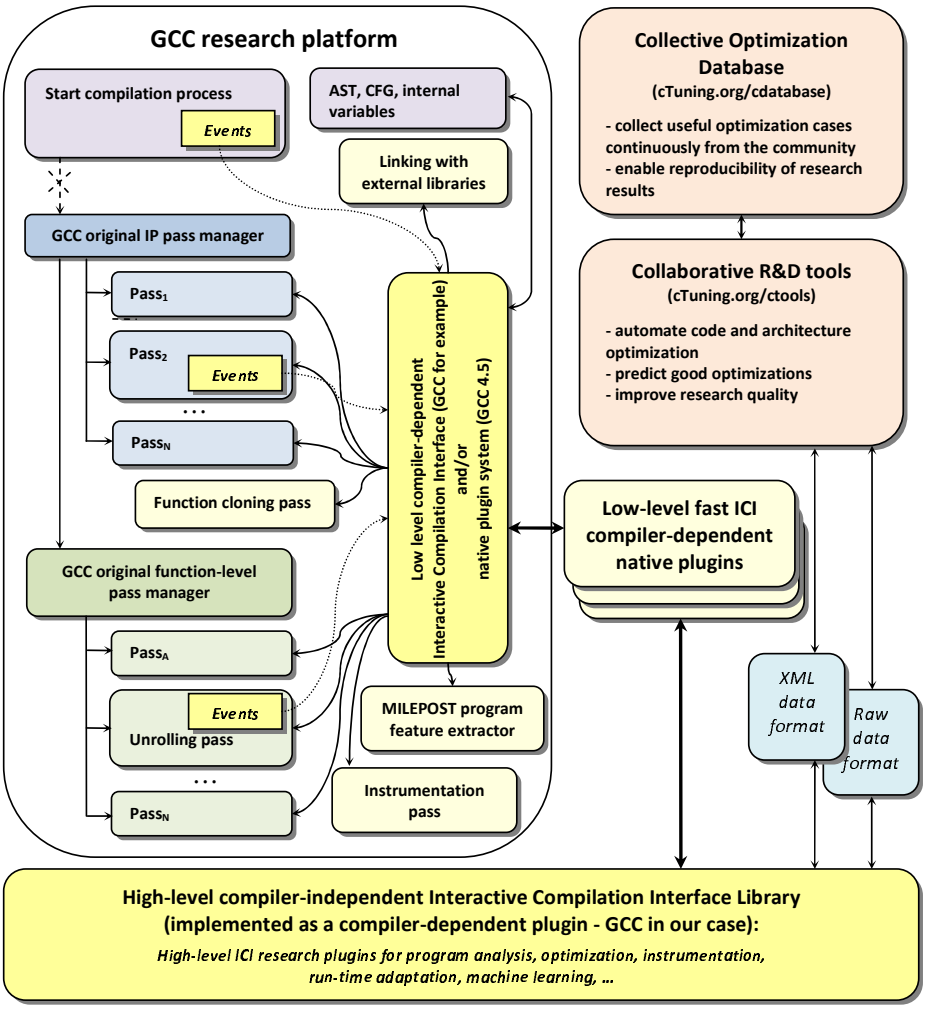


Fig. 1. GCC with high-level compiler-independent and low-level compiler-dependent ICI, and plugins as a research infrastructure connected with cTuning R&D tools and collective optimization database.

We pursue two main goals trying to transform GCC into a research compiler. First of all, we would like to have a common stable extensible compiler infrastructure shared by both academia and industry to improve the quality, practicality and reproducibility of research, and make experimental results immediately useful to the community. Second, we share the long-term vision of

the future adaptive self-tuning computing systems with the cTuning community [43] and therefore continue adding new functionality to GCC with ICI to enable statistical transparent collective optimization [29, 30]. The new ICI functionality provides the ability to substitute the compiler inter-procedural and function-level pass manager with arbitrary sequences of passes. It also includes new passes for generic function-level (and loop-level in the future) cloning, program instrumentation and MILEPOST extractor of static program features [19]. It is now possible to control some transformations such as unrolling individually. Finally, we added support for the XML data format in our GCC plugins to standardize and simplify communication with cTuning tools. Figure 1 shows the extended GCC with low-level compiler-dependent ICI (or native compiler plugin system) and a high-level compiler-independent ICI (that itself is implemented as a low-level plugin/library) connected to the cTuning optimization framework.

3 GSoC'09 extensions to GCC and ICI

This section describes our latest ICI extensions. All sources, plugins, implementation details and experimental results are available at the following collaborative development pages:

- http://ctuning.org/wiki/index.php/CTools:ICI:Projects:GSOC09:Fine_grain_tuning
- http://ctuning.org/wiki/index.php/CTools:ICI:Projects:GSOC09:Function_cloning_and_program_instrumentation

3.1 Enabling full control of GCC passes (selection and reordering)

One of the big problems researchers often face when using GCC is a constantly changing list of passes from one version to another, making their research tools dependent on a specific version of the compiler. We solve this problem by adding new functionality in ICI to obtain a list of available/executing passes and thus make research plugins more portable.

In GCC, all passes are invoked using `execute_one_pass` function. This function tests the pass gate status first and only then executes a pass itself. We added an ICI event call just before this test to send the name of the pass, its parameters and the gate status value to plugins. We wrote a plugin that works in a `record mode` and saves all passes with their original order, parameters and status of the gate.

GCC 4.4.x passes are stored in three linked lists: `all_lowering_passes`, `all_ipa_passes`, and `all_optimization_passes` (the latter is a misnomer). GCC 4.5.x has split the list `all_ipa_passes` into `all_small_ipa_passes` and `all_regular_ipa_passes`, and added `all_lto_gen_passes`. One should note that some of the intra-procedural passes can have function-level sub-passes, so we had to add extra functionality to be able to handle such situations in ICI. Finally, we added ICI event calls before each of these groups and provided a facility to skip execution of those groups of passes in GCC if triggered by the plugin. In

this case, a plugin written in a `reuse mode` can feed all passes back to a compiler in an arbitrary order and also execute auxiliary passes (such as function cloning, program instrumentation or feature extractor) on demand, thus gaining full control of the previously closed and hardwired compilation process.

In the last decade, multiple research projects have been investigating the selection of an optimal order of optimization passes [2, 6] using in-house research compilers. Now, we have a possibility to enhance these studies with a production compiler in a realistic environment but we face a new problem. Since GCC has not been designed for research, it provides very little information about dependencies between passes. This means that we can not explore a large search space of arbitrary orders of GCC passes due to frequent compiler crashes or invalid generated binaries.

(PASS_GROUP=) {*}PASS_GROUP1 { |INDIVIDUAL_PASSA(DEPENDENCE) {,INDIVIDUAL_PASSB(DEPENDENCE){,...}}{&INDIVIDUAL_PASSC(FORBID) {,INDIVIDUAL_PASSD(FORBID)}}}; {*}PASS_GROUP2...; {*}PASS_GROUP3

- PASS_GROUP can be a combination of other pass groups and individual passes;
- "*" means that a group from the right of this sign can be omitted; without it the group will always be selected (for initialization passes, etc).
- "|" means that a group from the left can be selected only if all groups from the right of this sign (separated by ",") have been also selected (true-dependence).
- "&" means that a group from the left can be selected only if none of the all groups from the right of this sign has been previously selected (anti-dependence).
- ";" means that the groups separated by this sign can be selected in any order.

Fig. 2. Formal definition of dependency grammar for GCC passes.

Therefore, we decided to develop a simple dependency grammar to be able to describe and generate valid sequences of optimization passes as described in Figure 2. In the future we plan to represent this grammar in the EBNF (Extended BackusNaur Form) [44], but for simplicity reasons now we use it as is it is presented in Fig. 2. We expect to provide a list of groups of passes with dependencies for each release of a compiler. Now, we have an ability to either generate the default order of passes as in GCC if we turn on all the passes from the list or generate an arbitrary valid sequence of passes for empirical performance/code size/compilation time exploration of various orders. Unfortunately, creating such a list of dependencies based on this grammar is a non-trivial task itself. It is an on-going work and we use both manual and automatic methods to find such dependencies. We start from the default order in GCC and start swapping passes each time checking that the compilation completed successfully and the code produced correct output on a number of datasets thus gradually finding dependencies between passes. We then verify each dependency manually.

Such methodology and grammar can in turn help to modularize GCC and test its correctness (semi-)automatically. We expect to build the first list of passes

with their dependencies for GCC 4.4.x within the next few months. Interestingly, we discovered an explicit dependency between pass "alias", which performs may-alias optimization, and pass "fre", which performs full redundancy elimination: placing "alias" pass after "fre" in some cases could lead to the changes in program semantic and consequently to the errors in produced binary program. In other words, "alias" should be placed always before"fre" lest the compiler could produce invalid code.

3.2 Enabling control of individual transformations

Control over selection of passes and their orders in GCC already opens up many research opportunities. However, our ultimate goal is to provide control over each individual transformation. Previously, special source to source tools have been used to optimize math libraries [1, 4, 8] and large applications [5] using iterative compilation with transformations such as loop tiling, interchange, unrolling and array padding among many others. Most of these transformations are now available inside GCC and other production compilers making them perfect candidates to substitute all specialized tuners.

We patch optimization passes to include event calls just before an individual transformations are applied. We pass all preceding information (decision to apply the transformation based on GCC optimization heuristic including its features and suggested parameters) to a plugin that can either just record this information for further off-line analysis (including machine learning techniques to learn good optimizations) or change the decision and parameters and force the compiler to change its internal decision.

Handling events for each transformation may sometimes slow down the compiler. There can be several solutions to that. We propose including both patched and non-patched optimization passes that can be controlled globally to control individual transformations only on parts of the code where that may have a high payoff in terms of performance or other benefits. We can also create self-adjusting passes that can register/remove events on demand.

We extend ICI to support handling of event parameters. ICI event parameters are actually pointers to temporal data that can live across several events before they are explicitly unregistered. When an event is issued, corresponding handler functions are executed and can read or write event parameters.

With the new ICI, it is fairly easy for researchers to record or reuse parameters of several common data types, such as integer. Pointers can also be registered in ICI as a parameter with the only difference that users have the responsibility to handle the type information correctly.

Together with the control of global optimization passes, the fine-grain control of transformations provides the ultimate control over full compiler optimization heuristic, opening up multiple research opportunities. We currently have support for loop unrolling and loop interchange (from GRAPHITE) and hope to provide support for the rest of transformations together with the community shortly.

3.3 Adding generic function cloning and program instrumentation

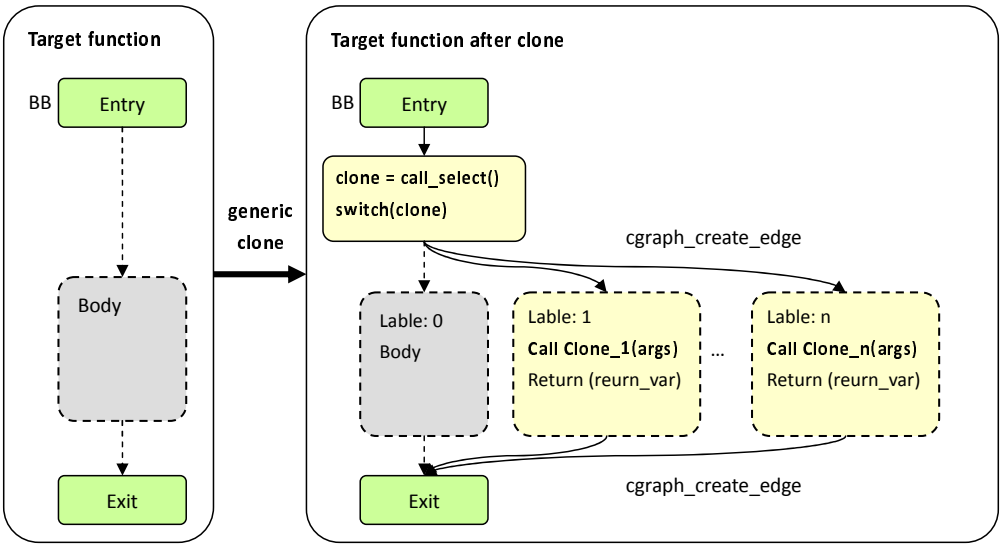


Fig. 3. Generic function cloning pass in GCC.

Multi-versioning helps to make static programs adaptive at run-time. It can be used to enable collective optimization, speed up iterative compilation by evaluating differently optimized versions at run-time and create self-tuning binaries adaptable to different inputs or architectures [25, 30, 29].

Therefore, we have implemented a new pass in GCC, named `generic_clone`, which can generate multiple copies of a given function, and insert a selection function at the beginning of the original function automatically as shown in Figure 3. To enable transparent modification of code (useful for collective optimization), we also add linking with external libraries without Makefile and GCC command-line modifications. These libraries may include different clone selection mechanisms for multiple practical and research purposes. For example, we are porting a clone selection mechanism from [25] to select differently optimized clones using hardware counters to enable adaptation of statically-compiled code for different program and system behavior at run-time. The call to the external selection function is followed by a switch structure to invoke selected clone.

We also have developed an instrumentation pass to be able to modify programs using plugins as shown in Figure 4. Currently, this pass can insert function calls to externally linked libraries at the beginning and the end of the compiled program to support collection of profile information for research tools, collective optimizers and self-tuning programs. We can also add such calls for any function including generated clones. This may be needed to monitor the behavior of the functions using external hardware counter libraries or connect program with

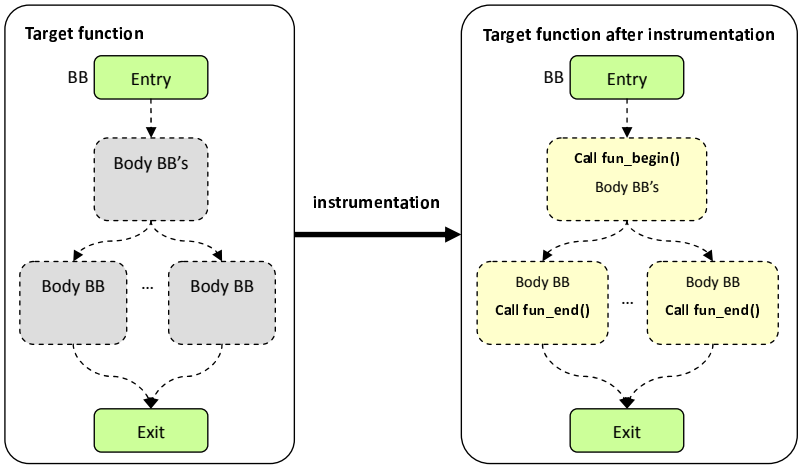


Fig. 4. Program instrumentation pass in GCC.

architecture simulators, etc. Importantly, we can instrument programs through plugins without any modifications to the source code thus keeping programs portable, simplifying development of program analyzers and enabling quick prototyping of research ideas. Eventually, we would like to provide program instrumentation capabilities on loop and even instruction level making GCC a powerful research tool for program analysis.

Both the generic cloning pass and instrumentation pass are implemented as `SIMPLE_IPA_PASS` in GCC and can be executed after the function availability is determined by the `visibility` pass. We added command line options to perform function cloning and instrumentation but we strongly recommend using ICI plugins to invoke these passes while avoiding modifying Makefiles or compilation scripts.

New extensions to GCC make it a powerful toolset to create adaptive binaries and libraries by combing cloning and instrumentation passes with the control of individual transformations to produce clones tuned at fine-grain level through external plugins.

3.4 Adding XML support for plugins data exchange

In the new ICI we decided to add XML support for data exchange between plugins and other tools besides simple row data format. We have several reasons to use XML:

- The format of the data exchanged between plugins and other tools is a well-structured and supports hierarchy.
- The XML format is widely used and can help users utilize their favorite tools to analyze the data.

```
<pass pass_name=''generic_cloning'' pass_type=''SIMPLE_IPA''>
  <function function_filename=''susan.c'' function_name=''susan_thin''>
   <clones >1</clones>
   <clone_name_extension >_clone</clone_name_extension>
   <adaptation_function >clone_select</adaptation_function>
   <options_clone >-O3</options_clone>
  </function>
</pass>
```

Fig. 5. Example of XML data file to perform function cloning using plugins.

```
<pass pass_name=''instrumentation'' pass_type=''SIMPLE IPA''>
  <function function_filename=''susan.c'' function_name=''susan_edges_small''
cloned=''1''>
   <add_function_call_before_func >_instr_start</add_function_call_before_func>
   <add_function_call_after_func >_instr_end</add_function_call_after_func>
  </function>
</pass>
```

Fig. 6. Example of XML data file to perform program instrumentation using plugins.

- The XML format is highly extensible, which is critical for future developments and backward compatibility.
- The XML format can be easily verified for correctness.

Figure 5 shows an example of the configuration file for a function cloning plugin in XML format. In this case, function 'susan_thin' from file 'susan.c' will be cloned once; a clone will be called `susan_thin_clone_1`; a selection function will be inserted calling `clone_select` and -O3 global optimization flag will be applied to a clone.

Figure 6 shows another example for function instrumentation. Function `susan_edges_small` from file 'susan.c' will be instrumented and additional function calls `susan_edges_small_instr_start` and `susan_edges_small_instr_end` will be inserted at the beginning and end of this function.

We are currently synchronizing the XML format for fine-grain program optimizations with the cTuning Collective Optimization Database format [45] to be able to store new experimental data there.

4 Experiments

In order to demonstrate our new research extensions to GCC and show their practicality, we perform several preliminary experiments on program optimization and adaptation (we plan to continue systematic experimentation in future work). For this preliminary study we decided to use both small kernels such as matrix multiply and a few larger applications from the MiBench/cBench benchmark suite [46].

We selected the following popular servers for our experiments:

- Dual-Core AMD Opteron 8218 with Red Hat Enterprise Linux AS release 4 X64_64 (referred later as Opteron machine, cTuning PLATFORM_ID = 11930834698757062);
- Intel Xeon E3110 running CentOS release 5.3, X86_64 (referred later as Xeon machine, cTuning PLATFORM_ID = 395021328416545100, ENVIRONMENT_ID = 7880645273825986);
- Intel Core2Duo T8300 running Linux Ubuntu SMP (referred later as Intel Core2Duo machine, cTuning PLATFORM_ID = 16563583955227076, ENVIRONMENT_ID = 42866903217278407);

Since we are still working on synchronizing our recent developments with mainline GCC, we performed our experiments using GCC 4.4.0 (cTuning COMPILER_ID = 129504539516446542) patched with ICI 2.0 and GSoC'09 extensions. We used the PAPI library [47] to obtain cycle accurate timing of our programs.

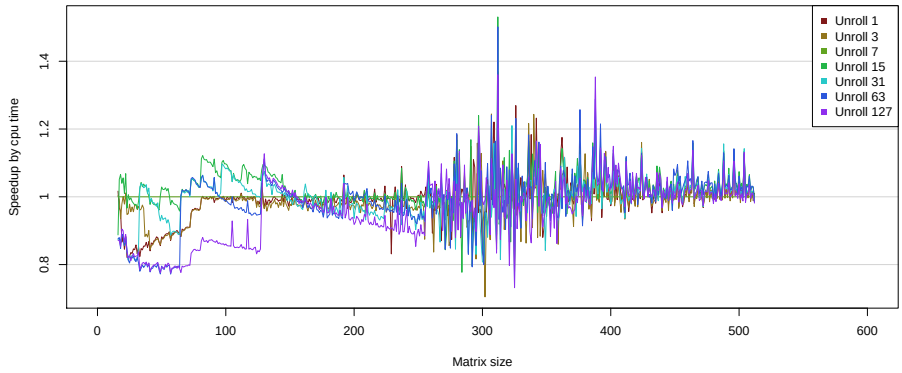
We provide cTuning unique IDs to help reviewers, readers and users verify and reproduce some of our results using cTuning Collective Optimization Database [45]. We hope that the dissemination of experimental results using common R&D tools and optimization repository will become a norm in the future and will help to speed up and improve academic and industrial research.

4.1 Controlling fine-grain program transformations in GCC

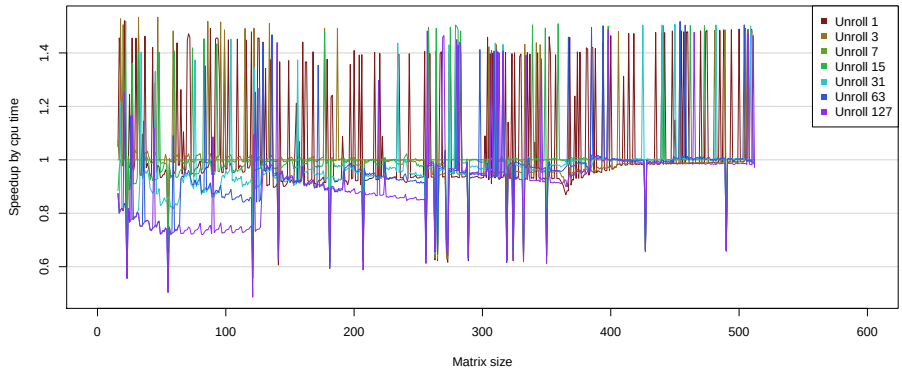
We decided to make a preliminary evaluation of the fine-grained control of transformations in GCC using a very simple and well-known example that up to now often needed specialized source-to-source tuners: optimizing matrix multiply using loop unrolling. Now, we can rely purely on a compiler to create and tune adaptive libraries.

We use the new instrumentation pass to add external cycle accurate timers from PAPI at the beginning and the end of the matrix multiply function. GCC's default unrolling heuristic suggest to unroll matmul 7 times when using `-O3 -funroll-loops` without taking data size into account. Since GCC 4.4 can only make power of two minus one copies of the loop body, i.e., unroll power of two times, we evaluated the following loop unrolling factors: 1, 3, 7, 31, 63 and 127 for square matrix sizes randing from 20x20 to 512x512.

The results from iterative compilation for Opteron and Xeon platforms are presented in Figure 7. They are similar to results obtained through source-to-source transformation from [1, 5, 48]. It clearly shows that the default static compiler optimization heuristic is incapable of producing the best code for a variety of inputs and fine-grain iterative compilation even with only loop unrolling can bring up to 1.5 times speedup. However, it can also bring considerable performance degradation for some combinations of datasets and unrolling numbers. The results for Opteron clearly show correlation of best unrolling factors, with the memory hierarchy showing complex interactions between various cache levels for large matrix sizes. Results for Intel are more difficult to explain and we leave detailed analysis for the future work. However, the results already show that



(a) AMD Opteron



(b) Intel Xeon

Fig. 7. Speedup of matmul for various matrix sizes when controlling loop unrolling in GCC through ICI.

GCC with the new ICI opens up many opportunities for research on fine-grain program optimizations, their interaction (particularly when adding more transformations including polyhedral optimizations) and performance prediction.

4.2 Creating adaptive programs and libraries

The experimental results from the previous section also motivate our static multiversioning approach in GCC to enable creation of adaptive applications. New extensions to ICI allow us to reproduce and extend the research framework from [25, 29, 38, 28] using GCC and select appropriately optimized functions based on the dataset and architecture features (using CPU ID and hardware counters). We will first replicate our technique to build an optimized run-time decision tree automatically using statistical and machine learning techniques as in [28].

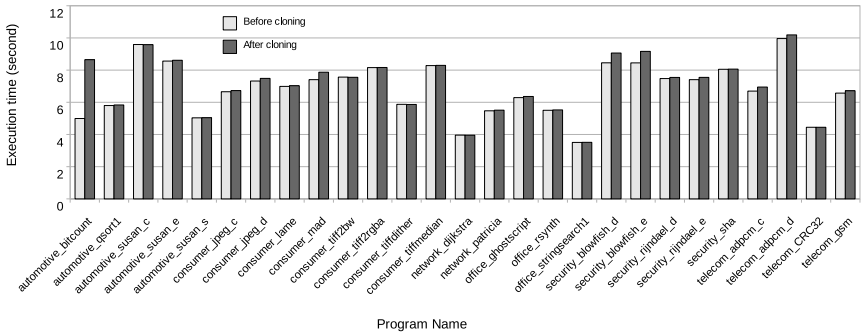


Fig. 8. Overhead of call-switch mechanism during generic function cloning.

Naturally, a run-time overhead may be introduced by our call-switch mechanism. We decided to perform preliminary experiments to evaluate this overhead using cBench benchmark. We selected 6 hot functions covering most of the execution time of all programs using OProfile (excluding `main`), cloned them once and added a cyclical selection mechanism using GCC with new ICI. Figure 8 clearly demonstrates that at least for MiBench/cBench, the execution time overhead of our call-switch mechanism is negligible in most of the cases in comparison with the original code. Figure 9 also shows the negligible growth of binaries after cloning all hot functions, which is critical for embedded systems. These results are similar to results from the [25] when using source-to-source cloning.

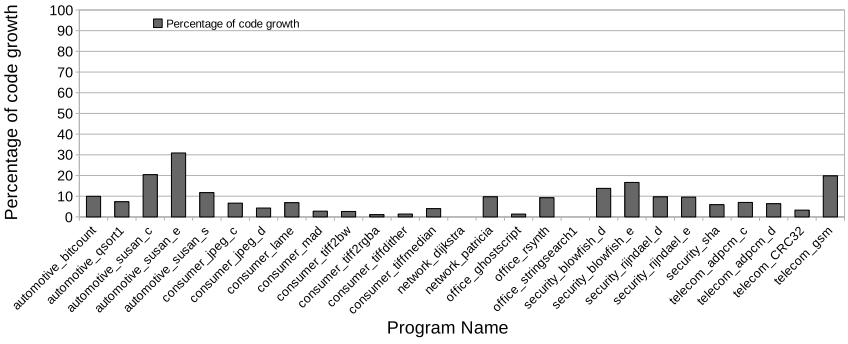


Fig. 9. Binary growth when cloning all hot functions once.

Best flags taken from cTuning database [45] vs baseline(-O3)	susan_e	dijkstra	sha_e
... alias retslot ... crited sink loop ... loopdone vrp ...	28% 762541000430973173	5% 841507490430918931	26% 130797385743093369
... sink **alias** loop ... loopdone vrp ...	34% 127297480343098038	8% 208941853843093041	29% 149436739843093444
... sink **alias** loop ... loopdone **retslot** ... **crited** vrp...	28% 193582669430976651	9% 576051282430931424	31% 350720421430934188

Table 1. Speedups and associated cTuning RUN_ID (to reproduce results if needed) over -O3 for preliminary manual pass reordering experiments using three cBench programs, MILEPOST GCC 4.4.0 with new ICI extensions and Intel Core2Duo machine.

4.3 Preliminary evaluation of pass reordering

Since we have now enabled the optional arbitrary pass selection and ordering in GCC, we would like to evaluate potential performance improvements from different pass sequences. We tried first to reproduce some of the results from [2, 6] but have not succeeded so far. We spent some time manually reordering passes in the "all_optimizations" group and finally found several programs where different pass orders improve the code over the default GCC optimization order and the best selection of optimization passes/flags from cTuning repository [45]. Table 1 shows how the position of pass "alias","retslot" and "crited" influenced overall speedup over -O3, the best default GCC optimization heuristic. Moreover, the most profitable pass sequences depend on the program being optimized. Though this dependence is not yet large, it still shows new research opportunities in GCC and motivates us to extend research from [19] and learn good optimization orders for a given program, architecture and a dataset using statistical and machine learning techniques.

5 Synchronization with mainline GCC 4.5

We have spent more than 3 years on ICI developments and this framework has become too complex to patch for each new release. Since we hope that it can be eventually useful for both research community and GCC end-users, we would naturally like now to move it to mainline GCC. After many discussions, the forthcoming release of GCC will finally feature a low-level plugin framework. However, it is still quite different from ICI. For example, ICI has been written to allow researchers to insert new code easily in random places within GCC without much planning: events and parameters have names and are managed in a hash table which is easy to deal with but may have performance overhead at each event raising site even if there is no callback for that event. In contrast, GCC 4.5 plugins have been designed to have a very low overhead, but require explicitly adding an enum number and a name for every new event, and all parameters have to be passed via a single pointer which may potentially result in many ad-hoc structs. We will try to address this by having a special wrapper to pass a list of named parameters using a va_list* through the original GCC 4.5 events interface. We will also have a number of pre-existing events in GCC 4.5 which we

may want to interface with the ICI named parameters. We also plan to discuss with the GCC community whether the ICI type description could be accepted in GCC. Finally, we separated ICI into high-level compiler-independent research interface and a low-level compiler-dependent fast low-level interface synchronized with the native plugin framework in GCC 4.5.

6 Conclusions and Future Work

In this article we presented our recent GSoC'09 extensions to GCC plugin system to simplify and popularize the use of this free, wide-spread open-source compiler in realistic research on code and architecture optimization. The new infrastructure separates ICI into high-level compiler-independent and low-level compiler-dependent libraries and provides support for generic function cloning and run-time adaptation for statically-compiled programs in heterogeneous environments, inter-procedural and function-level optimization pass selection and reordering with a dependency grammar able to describe valid sequences, control of individual transformations and their parameters for fine-grain application optimization, and the XML representation of the compilation flow to ease communication with external tools.

We are currently synchronizing the low-level Interactive Compilation Interface and GSoC'09 extensions with mainline GCC and its new native plugin framework to provide a reasonably stable compiler-independent API to the research community during rapid compiler evolution. We will be gradually adding external control of OpenMP and individual transformations including inlining, vectorization and polyhedral loop transformations from the GRAPHITE pass. We plan to provide support for program instrumentation and instruction manipulation for advanced code analysis and optimization. Eventually, researchers would also like to have source to source transformations in GCC as well as support for dynamic optimization and split compilation (using MONO and GCC4CIL, for example), remove hard-coded dependencies between passes, and exploit direct access to global variables. Finally, we would like to start systematic investigation of the correctness of automatically generated combinations of optimizations. This is of particular importance during statistical collective optimization [29] when using the cTuning framework with GCC [30, 43] for embedded devices, data centers and cloud computing systems for automatic, continuous and transparent performance/power tuning of user applications or for whole system optimization (such as Moblin and Android).

7 Acknowledgments

Yuanjie Huang and Liang Peng have been supported by Google Summer of Code program'09 program to implement fine-grain tuning, function cloning and program instrumentation. Yuriy Kashnikov has been supported by UVSQ to implement pass reordering in GCC. Joern Renneke has been supported by INRIA to move the Interactive Compilation Interface to mainline GCC and synchronize it with the current GCC 4.5

plugin system. We would like to thank multiple users from GCC, cTuning and HiPEAC communities for their useful feedback. We would also like to thank Prof. William Jalby for interesting discussions about ICI and program optimizations. Finally, we would like to thank anonymous reviewers, Jeremmy Bennett and Phil Barnard for their insightful comments to improve this article.

References

1. Whaley, R., Dongarra, J.: Automatically tuned linear algebra software. In: Proceedings of the Conference on High Performance Networking and Computing. (1998)
2. Cooper, K., Schielke, P., Subramanian, D.: Optimizing for reduced code space using genetic algorithms. In: Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES). (1999) 1–9
3. Bodin, F., Kisuki, T., Knijnenburg, P., O'Boyle, M., Rohou, E.: Iterative compilation in a non-linear optimisation space. In: Proceedings of the Workshop on Profile and Feedback Directed Compilation. (1998)
4. Matteo, F., Johnson, S.: FFTW: An adaptive software architecture for the FFT. In: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing. Volume 3., Seattle, WA (May 1998) 1381–1384
5. Fursin, G., O'Boyle, M., Knijnenburg, P.: Evaluating iterative compilation. In: Proceedings of the Workshop on Languages and Compilers for Parallel Computers (LCPC). (2002) 305–315
6. Kulkarni, P., Zhao, W., Moon, H., Cho, K., Whalley, D., Davidson, J., Bailey, M., Paek, Y., Gallivan, K.: Finding effective optimization phase sequences. In: Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES). (2003) 12–23
7. Triantafyllis, S., Vachharajani, M., Vachharajani, N., August, D.: Compiler optimization-space exploration. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO). (2003) 204–215
8. Singer, B., Veloso, M.: Learning to predict performance from formula modeling and training data. In: Proceedings of the Conference on Machine Learning. (2000)
9. Pan, Z., Eigenmann, R.: Fast and effective orchestration of compiler optimizations for automatic performance tuning. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO). (2006) 319–332
10. Heydemann, K., Bodin, F.: Iterative compilation for two antagonistic criteria: Application to code size and performance. In: Proceedings of the 4th Workshop on Optimizations for DSP and Embedded Systems, colocated with CGO. (2006)
11. Hoste, K., Eeckhout, L.: Cole: Compiler optimization level exploration. In: Proceedings of International Symposium on Code Generation and Optimization (CGO). (2008)
12. Nisbet, A.: GAPS: Genetic algorithm optimised parallelization. In: Proceedings of the Workshop on Profile and Feedback Directed Compilation in conjunction with PACT'98. (1998)
13. : ACOVEA: Using Natural Selection to Investigate Software Complexities. `http://www.coyotegulch.com/products/acovea`
14. : Learning to schedule straight-line code. In: Proceedings of the Conference on Neural Information Processing Systems (NIPS). (1997)

15. Monsifrot, A., Bodin, F., Quiniou, R.: A machine learning approach to automatic production of compiler heuristics. In: Proceedings of the International Conference on Artificial Intelligence: Methodology, Systems, Applications. LNCS 2443 (2002) 41–50
16. Stephenson, M., Martin, M., O'Reilly, U.: Meta optimization: Improving compiler heuristics with machine learning. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). (2003) 77–90
17. Stephenson, M., Amarasinghe, S.: Predicting unroll factors using supervised classification. In: Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO). (2005)
18. Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M., Thomson, J., Toussaint, M., Williams, C.: Using machine learning to focus iterative optimization. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO). (2006)
19. Fursin, G., Miranda, C., Temam, O., Namolaru, M., Yom-Tov, E., Zaks, A., Mendelson, B., Barnard, P., Ashton, E., Courtois, E., Bodin, F., Bonilla, E., Thomson, J., Leather, H., Williams, C., O'Boyle, M.: Milepost gcc: machine learning based research compiler. In: Proceedings of the GCC Developers' Summit. (June 2008)
20. Dubach, C., Jones, T.M., Bonilla, E.V., Fursin, G., O'Boyle, M.F.: Portable compiler optimization across embedded programs and microarchitectures using machine learning. In: Proceedings of the 42nd International Symposium on Microarchitecture (MICRO). (December 2009)
21. Tournavitis, G., Wang, Z., Franke, B., O'Boyle, M.F.: Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI). (2009)
22. Voss, M., Eigenmann, R.: Adapt: Automated de-coupled adaptive program transformation. In: Proceedings of the International Conference on Parallel Processing (ICPP). (2000)
23. Lu, J., Chen, H., Yew, P.C., Hsu, W.C.: Design and implementation of a lightweight dynamic optimization system. In: Journal of Instruction-Level Parallelism. Volume 6. (2004)
24. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO), Palo Alto, California (March 2004)
25. Fursin, G., Cohen, A., O'Boyle, M., Temam, O.: A practical method for quickly evaluating program optimizations. In: Proceedings of the 1st International Conference on High Performance Embedded Architectures & Compilers (HiPEAC). Number 3793 in LNCS, Springer Verlag (November 2005) 29–46
26. Stephenson, M.W.: Automating the Construction of Compiler Heuristics Using Machine Learning. PhD thesis, MIT, USA (2006)
27. Lau, J., Arnold, M., Hind, M., Calder, B.: Online performance auditing: Using hot optimizations without getting burned. In: Proceedings of the ACM SIGPLAN Conference on Programming Languaged Design and Implementation (PLDI). (2006)
28. Luo, L., Chen, Y., Wu, C., Long, S., Fursin, G.: Finding representative sets of optimizations for adaptive multiversioning applications. In: 3rd Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation (SMART'09), colocated with HiPEAC'09 conference. (January 2009)

29. Fursin, G., Temam, O.: Collective optimization. In: Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2009). (January 2009)
30. Fursin, G.: Collective tuning initiative: automating and accelerating development and optimization of computing systems. In: Proceedings of the GCC Developers' Summit. (June 2009)
31. Wilson, R.P., French, R.S., Wilson, C.S., Amarasinghe, S.P., Anderson, J.A.M., Tjiang, S.W.K., Liao, S.W., Tseng, C.W., Hall, M.W., Lam, M.S., Hennessy, J.L.: Suif: An infrastructure for research on parallelizing and optimizing compilers. SIGPLAN Notices **29**(12) (1994) 31–37
32. : ROSE Compiler Infrastructure. `http://www.rosecompiler.org`
33. : LLVM Compiler Infrastructure. `http://llvm.org`
34. : GNU Compiler Collection. `http://gcc.gnu.org`
35. Cornero, M., Costa, R., Pascual, R.F., Ornstein, A.C., Rohou, E.: An experimental environment validating the suitability of cli as an effective deployment format for embedded systems. In: Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC). (January 2008)
36. : MONO: cross platform, open source .NET development framework. `http://www.mono-project.com`
37. : ICI: Interactive Compilation Interface: plugin system to convert production compilers into research toolsets (2005)
38. Jimenez, V., Gelado, I., Vilanova, L., Gil, M., Fursin, G., Navarro, N.: Predictive runtime code scheduling for heterogeneous architectures. In: Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2009). (January 2009)
39. Long, S., Fursin, G., Franke, B.: A cost-aware parallel workload allocation approach based on machine learning techniques. In: Proceedings of the IFIP International Conference on Network and Parallel Computing (NPC 2007). Number 4672 in LNCS, Springer Verlag (September 2007) 506–515
40. Starynkevitch, B.: Multi-stage construction of a global static analyser. In: GCC Developers' Summit. (July 2007)
41. Glek, T., Mandelin, D.: Using gcc instead of grep and sed. In: Proceedings of the GCC Developers' Summit. (June 2008)
42. Sean Callanan, D.D., Zadok, E.: Extending gcc with modular gimple optimizations. In: GCC Developers' Summit. (July 2007)
43. : cTuning.org: Collective tuning center to automate design and optimization of computing systems. `http://cTuning.org` (2008)
44. Whitney, G.: An extended bnf for specifying the syntax of declarations. In: AFIPS '69 (Spring): Proceedings of the May 14-16, 1969, spring joint computer conference, New York, NY, USA, ACM (1969) 801–812
45. : cTuning optimization repository (Collective Optimization Database). `http://ctuning.org/cdatabase`
46. : Collective Benchmark: collection of open-source programs and multiple datasets from the community. `http://ctuning.org/cbench`
47. : PAPI: A Portable Interface to Hardware Performance Counters. `http://icl.cs.utk.edu/papi`
48. Fursin, G.: Iterative Compilation and Performance Prediction for Numerical Applications. PhD thesis, University of Edinburgh, United Kingdom (2004)