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.

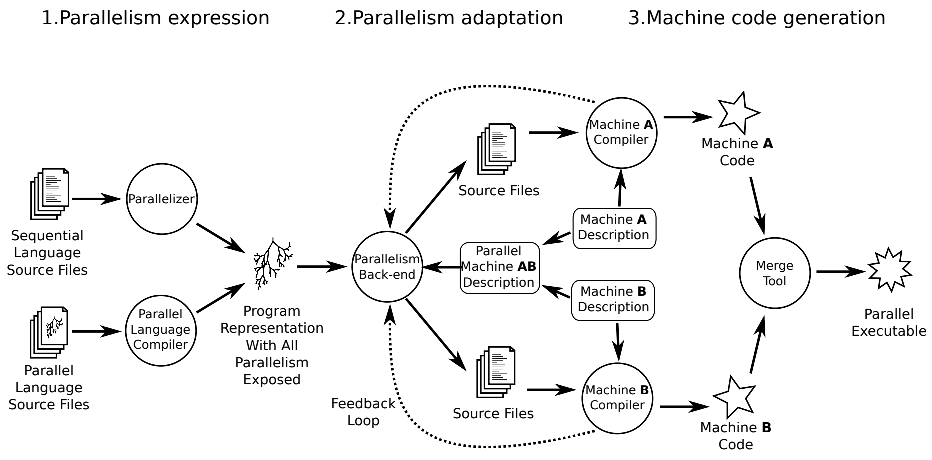


Fig. 1. Compilation flow integrating parallelism adaptation.

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

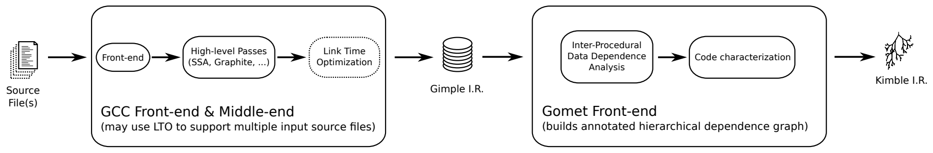

Source File(s)
Front-end
High-level Passes (SSA, Graphite, ...)
Link Time Optimization
GCC Front-end & Middle-end
(may use LTO to support multiple input source files)
Gimple I.R.
Inter-Procedural Data Dependence Analysis
Code characterization
Gomet Front-end
(builds annotated hierarchical dependence graph)
Kimble I.R.

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).

$$Nodes = \{Function, Loop, Region, Cluster, Statement, Call\}$$

$$\left.\begin{array}{rl} Tree_N \rightarrow & (Tree_N) \\ | & Tree_N \parallel Tree_N \\ | & Tree_N ; Tree_N \\ | & n \in N \end{array}\right\} \text{Dependence graph expression}$$

$$\left.\begin{array}{ll} Function & \rightarrow Function(Tree_{\{Loop, Region\}}) \\ Loop & \rightarrow Loop(Tree_{\{Loop, Region\}}) \\ Region & \rightarrow Region(Tree_{\{Cluster, Statement, Call\}}) \\ Call & \rightarrow Call(Tree_{\{Function\}}) \end{array}\right\} \text{Hierarchy expression}$$

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.

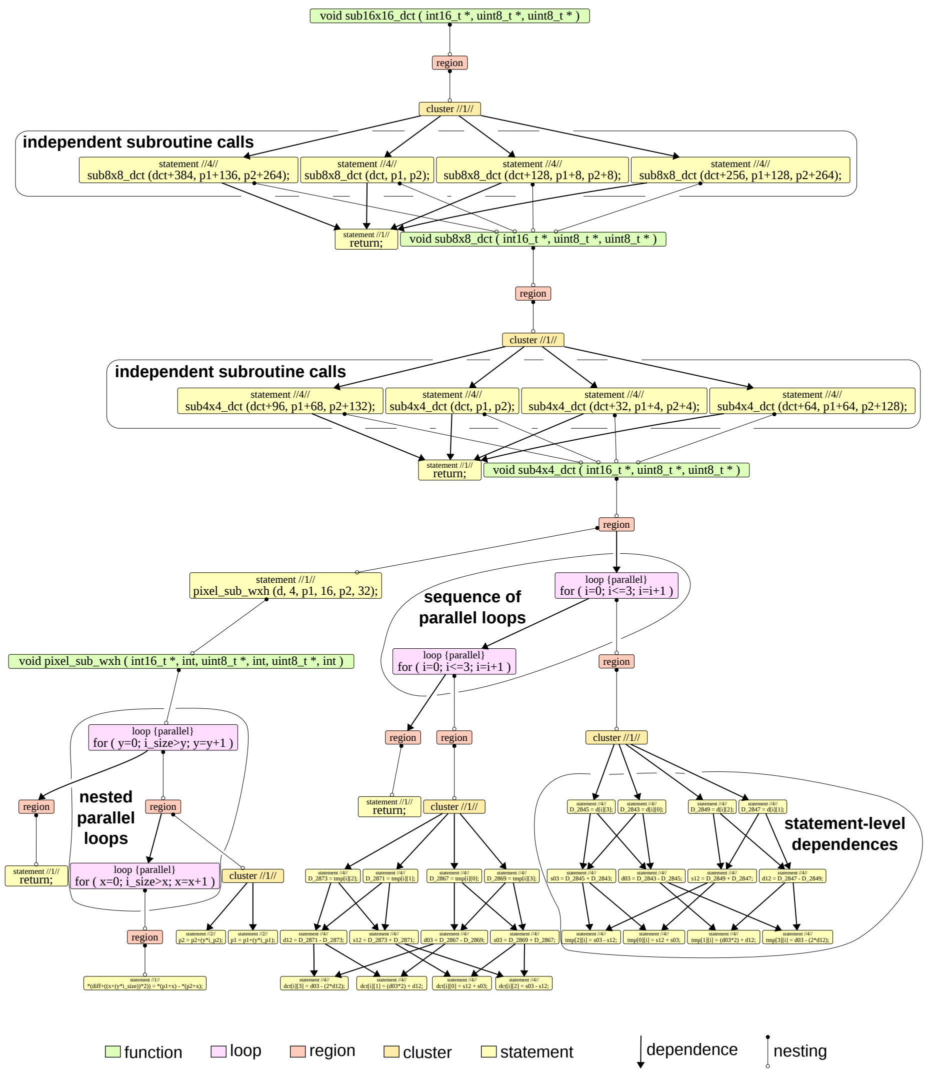
As the grammar ruling Kimble suggests, the parallelism information expressed directly map to C constructs. This eases parallelism adaptation as this information is conserved and transparently updated during program transformation.

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



void sub16x16_dct (int16_t *, uint8_t *, uint8_t *)
region
cluster //1//
independent subroutine calls
statement //4// sub8x8_dct (dct+384, p1+136, p2+264);
statement //4// sub8x8_dct (dct, p1, p2);
statement //4// sub8x8_dct (dct+128, p1+8, p2+8);
statement //4// sub8x8_dct (dct+256, p1+128, p2+264);
statement //1// return;
void sub8x8_dct (int16_t *, uint8_t *, uint8_t *)
region
cluster //1//
independent subroutine calls
statement //4// sub4x4_dct (dct+96, p1+68, p2+132);
statement //4// sub4x4_dct (dct, p1, p2);
statement //4// sub4x4_dct (dct+32, p1+4, p2+4);
statement //4// sub4x4_dct (dct+64, p1+64, p2+128);
statement //1// return;
void sub4x4_dct (int16_t *, uint8_t *, uint8_t *)
region
statement //1// pixel_sub_wxh (d, 4, p1, 16, p2, 32);
loop {parallel} for (i=0; i<=3; i=i+1)
sequence of parallel loops
loop {parallel} for (i=0; i<=3; i=i+1)
void pixel_sub_wxh (int16_t *, int, uint8_t *, int, uint8_t *, int)
loop {parallel} for (y=0; i_size>y; y=y+1)
nested parallel loops
loop {parallel} for (x=0; i_size>x; x=x+1)
cluster //1//
statement //1// return;
statement //2// p2 = p2+(y*i_p2);
statement //2// p1 = p1+(y*i_p1);
statement //1// *(diff+((x+(y*i_size))*2)) = *(p1+x) - *(p2+x);
statement //1// return;
cluster //1//
statement //4// D_2873 = tmp[i][2];
statement //4// D_2871 = tmp[i][1];
statement //4// D_2867 = tmp[i][0];
statement //4// D_2869 = tmp[i][3];
statement //4// d12 = D_2871 - D_2873;
statement //4// s12 = D_2873 + D_2871;
statement //4// d03 = D_2867 - D_2869;
statement //4// s03 = D_2869 + D_2867;
statement //4// dct[i][3] = d03 - (2*d12);
statement //4// dct[i][1] = (d03*2) + d12;
statement //4// dct[i][0] = s12 + s03;
statement //4// dct[i][2] = s03 - s12;
cluster //1//
statement //4// D_2845 = d[i][3];
statement //4// D_2843 = d[i][0];
statement //4// D_2849 = d[i][2];
statement //4// D_2847 = d[i][1];
statement-level dependences
statement //4// s03 = D_2845 + D_2843;
statement //4// d03 = D_2843 - D_2845;
statement //4// s12 = D_2849 + D_2847;
statement //4// d12 = D_2847 - D_2849;
statement //4// tmp[2][i] = s03 - s12;
statement //4// tmp[0][i] = s12 + s03;
statement //4// tmp[1][i] = (d03*2) + d12;
statement //4// tmp[3][i] = d03 - (2*d12);
🟩 function
🟪 loop
🟥 region
🟧 cluster
🟨 statement
dependence
nesting

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.

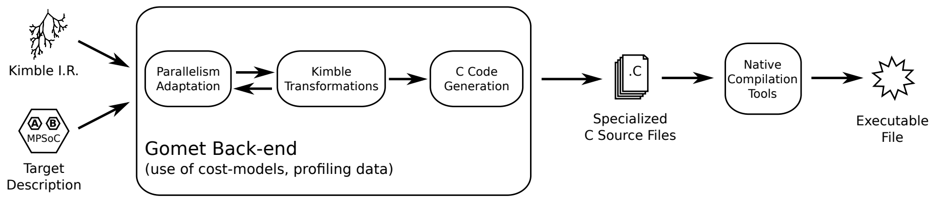


Fig. 5. Inputs and outputs of Gomet's back-end.

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

```
void sub4x4_dct ( int16_t *dct, uint8_t *p1, uint8_t *p2 )
{
  int16_t D_2820;
  int D_2821;
  [...]  /* more local declarations, including SSA variables */
  int16_t d[4][4];
  int d12;

  goto R_13;  /* control flow restitution with gotos and labels */
 R_13:
  d_1 = (int16_t *) &(d);
  pixel_sub_wxh ( d_1, 4, p1, 16, p2, 32 );
  goto L_2;
 L_2:
  for ( i=0; i<=3; i=i+1 )  /* reconstituted loop construct */
    {
      goto R_11;
 R_11:
      D_2820 = d[i][0];
      D_2821 = (int) D_2820;
      D_2822 = d[i][3];
      D_2823 = (int) D_2822;
      s03 = D_2823 + D_2821;
      [...]  /* more SSA  statements */
      tmp[3][i] = D_2843;
      goto R_12;
 R_12:
      goto L_2_;
 L_2_:
    ;
    }
  goto R_14;
 L_1:
  for ( i=0; i<=3; i=i+1 )  /* reconstituted loop construct */
    {
      goto R_9;
 R_9:
      D_2844 = tmp[i][0];
      [...]  /* more SSA  statements */
      D_2854 = (int16_t *) ((void *)dct + D_2853);
      D_2854[2] = D_2838;
      goto R_10;
 R_10:
      goto L_1_;
 L_1_:
    ;
    }
  goto R_15;
 R_15:
  return;
 R_14:
  goto L_1;
}
```

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

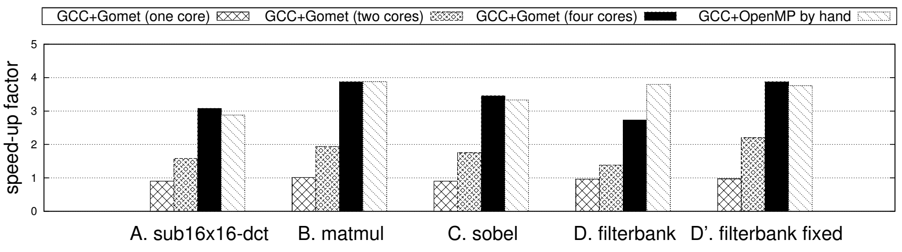


Fig. 7. Speed-up measurement of four programs processed with GCC+Gomet.

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

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

A. Original code

```
Vect_H_I_I_lsm_2 = Vect_H[i][j];
D_2798 = 1;
for ( k=0; D_2798!=0;  )
 {
   D_2785 = Vect_H_I_I_lsm_2;
   D_2788 = D_2787[k];
   D_2789 = j - k;
   D_2790 = (long unsigned int) D_2789;
   D_2791 = D_2790 * sizeof(double);
   D_2792 = (double *) ((void *)r + D_2791);
   D_2793 = *(D_2792);
   D_2794 = D_2788 * D_2793;
   D_2795 = D_2785 + D_2794;
   Vect_H_I_I_lsm_2 = D_2795;
   k = k + 1;
   D_2796 = k < N_col;
   D_2797 = k <= j;
   D_2798 = D_2796 && D_2797;
 }
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