Clint: A Direct Manipulation Tool for Parallelizing Compute-Intensive Program Parts
Oleksandr Zinenko, Stéphane Huot, Cédric Bastoul

To cite this version:
Oleksandr Zinenko, Stéphane Huot, Cédric Bastoul. Clint: A Direct Manipulation Tool for Parallelizing Compute-Intensive Program Parts. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Jul 2014, Melbourne, Australia. IEEE, 2014. <hal-01055788>

HAL Id: hal-01055788
https://hal.inria.fr/hal-01055788
Submitted on 13 Aug 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Clint: A Direct Manipulation Tool for Parallelizing Compute-Intensive Program Parts

Oleksandr Zinenko  
Inria, Université Paris-Sud & CNRS (LRI)  
Orsay, France  
Email: oleksandr.zinenko@inria.fr

Stéphane Huot  
Université Paris-Sud & CNRS (LRI), Inria  
Orsay, France  
Email: stephane.huot@lri.fr

Cédric Bastoul  
University of Strasbourg, Inria  
Strasbourg, France  
Email: cedric.bastoul@unistra.fr

Abstract—Parallel systems are now omnipresent and their effective use requires significant effort and expertise from software developers. Multitude of languages and libraries offer convenient ways to express parallelism, but fall short at helping programmers find parallelism in existing programs. To address this issue, we introduce Clint, a direct manipulation tool aimed to ease both the extraction and the expression of parallelism. Clint builds on polyhedral representation of programs to convey dynamic behavior, to perform automatic data dependence analysis and to ensure code correctness. It can be used to rework and improve automatically generated optimizations and to make manual program transformation faster, safer and more efficient.

Keywords: parallel programming, polyhedral model, software visualization, direct manipulation interface.

I. INTRODUCTION

The large scale adoption of parallel architectures in modern electronic devices urges the software industry to provide developers with tools for building efficient parallel applications. Despite tremendous effort, parallel program development remains challenging, especially when porting legacy sequential applications or targeting new parallel architectures. Several languages and libraries, such as Cilk, X10, Chapel or UPC, offer high-level parallelism expression constructs, but parallelism extraction is left under the responsibility of the developer.

Most existing approaches to automate parallelism extraction focus on compiler techniques for loop vectorization and thread-level parallelization and rely on an algebraic representation of programs called the polyhedral model [1]–[3]. It captures the dynamic behavior of a class of compute-intensive loops and allows to perform accurate data dependence analysis to ensure semantics preservation, and to restructure the input program to expose parallelism. However, polyhedral compilers are black-boxes that do not help users if the proposed restructuring does not solve their problem, e.g. has worse dynamic characteristics or does not expose parallelism due to the lack of semantics preservation hints. Semi-automatic tools based on directive scripts were proposed [4], [5], but this method has seen little interest as writing such scripts remains a tedious task, requiring expertise in a complex underlying theory.

Although the polyhedral model has a direct graphical representation featuring its geometric nature, it has not been explored yet as a way to leverage the power of the model through interactive visualizations. We have thus designed Clint, a tool to support parallelization through interactive direct manipulation of loop nest visualization (Fig. 1). It presents loop iteration spaces as geometric shapes, dynamic statement executions as solid dots and data dependences as arrows, together with continuous feedback during manipulation. Clint maintains correspondence between the valid source code and the visualization using polyhedral compilation techniques. This interactive approach may ease the design and exploration of program transformations comparing to manual or even semi-automatic code restructuring thanks to immediate feedback, editable history and undoability, allowing programmers to focus on parallelism given that code semantic equivalence is guaranteed by the tool.

II. POLYHEDRAL REPRESENTATION OF PROGRAMS

The polyhedral model is an algebraic representation of a subset of imperative programs that encodes dynamic executions of statements [1]. It was designed to represent static control parts (SCoPs), which are loops with affine control and memory accesses, i.e., such that conditions and array subscripts are linear forms of outer loop counters and constants. Even with these restrictions, the model captures a broad range of compute-intensive, execution time-consuming loops that can be optimized [6]. Moreover, extensions to the model make it applicable up to entire functions [7].

The model considers dynamic statement instances, which are modeled for each statement as integer points inside a polyhedron called the iteration domain of the statement, that captures loops and branches surrounding it. For the polynomial multiplication loop nest presented in Fig. 2a, it contains two-dimensional vectors where the first dimension corresponds to the outer loop on $i$ and the second to the inner loop on $j$. 
(Fig. 2b). Each vector refers to a particular execution of the statement $S$. Several compilers have the ability to raise SCoPs to a polyhedral form such as GNU GCC$^1$ or LLVM$^2$.

$$\begin{align*}
\text{for } (i = 0; i < N; i++) \\
\text{for } (j = 0; j < N; j++) \\
S: z[i+j] \leftarrow x[i] \cdot y[j];
\end{align*}$$

$$D_S(N) = \left\{ \left( \begin{array}{c} i \\ j \end{array} \right) \mid \begin{array}{l} 0 \leq i < N \\ 0 \leq j < N \end{array} \right\}$$

(a) Polynomial multiply kernel (b) Iteration domain of statement $S$

Fig. 2. Iteration domain of a statement.

The statement instances can be executed following a specific order defined by associating each point in the iteration domain with a logical execution date by means of a mapping relation. In case of parallel systems, multiple points having equal dates can be executed simultaneously given that a location of execution (e.g. processor core) is also associated with each instance. Mapping relations are a convenient and expressive way to define a complex composition of program transformations including, e.g., loop interchange, fusion, fission, skewing, tiling, index-set-splitting [5]. To ensure semantic equivalence, any mapping relation must not violate data dependences, i.e. preserve the relative order of instances accessing the same memory location.

A powerful property of the polyhedral approach is to make it possible to compute exact and instance-wise data dependences, and to check the mapping correctness against them [8]. It allows polyhedral compilers to construct correct mapping relations for various objectives automatically, e.g. to efficiently use cache memory or to expose parallelism. For the polynomial multiplication example, a mapping defined in Fig. 3a groups accesses to the same memory address $z[1+j]$ in a single iteration of the outer loop, removing the data dependence between iterations, and thus making this loop parallelizable. The mapping reshapes the iteration domain to the form of Fig. 3b.

$$\begin{align*}
\theta_S(N) = \left\{ \left( \begin{array}{c} i \\ j \end{array} \right) \rightarrow \left( \begin{array}{c} t_1 \\ t_2 \end{array} \right) \mid \begin{array}{l} t_1 = i + j \\ t_2 = i \end{array} \right\}
\end{align*}$$

(a) Example time mapping for the statement $S$ of Figure 2

$$\begin{align*}
\tau_S(N) = \left\{ \left( \begin{array}{c} t_1 \\ t_2 \end{array} \right) \mid \begin{array}{l} 0 \leq t_1 - t_2 < N \\ 0 \leq t_2 < N \end{array} \right\}
\end{align*}$$

(b) Transformed iteration domain $D_S(N)$ by time mapping $\theta_S(N)$

Fig. 3. Time mapping example and resulting transformed iteration domain.

Once the transformed domains have been computed in the model, the resulting code is generated by building a program that enumerates the integer points within the domain with respect to lexicographic ordering of their coordinates. Several efficient algorithms and tools exist for that purpose including CLooG [3] and CodeGen+ [9].

```c
#pragma omp parallel for private(t2)
for (t1 = 0; t1 <= 2*N-2; t1++)
  for (t2 = max(0, t1-N+1); t2 <= min(t1, N-1); t2++)
S: z[t1] += x[t2] * y[1-t2];
```

Fig. 4. Generated code from the transformed domain $\tau_S(N)$.

Fig. 4 shows the final code generated from the transformed domain in Fig. 3b with a parallel loop on $t_1$. The parallelizing optimization corresponds here to a classic loop transformation called skewing [6]. Although several automatic mapping algorithms were proposed, they are not necessarily resulting in the best parallelizing solution because of the multiple heuristics they use to handle complex underlying processor architectures [2], [3]. Furthermore, since they heavily transform the original program structure, the result is often unreadable for a programmer and thus uncorrectable. Semi-automatic approaches allow programmers to construct and modify a mapping relation freely, but they do not help in finding an appropriate transformation and still require significant expertise to describe it in a special syntax. Our objective with Clint is thus to provide programmers with an interactive tool to ease both parallelism extraction through direct manipulation of visual dependence abstraction and parallelism expression through automatic code generation.

III. THE CLINT INTERACTIVE CODE VISUALIZATION

Design rationale – Clint is a direct manipulation interface designed to (i) help programmers with parallelizing compute-intensive programs parts; (ii) ease the exploration of possible transformations; and (iii) guarantee the correctness of the final code. Clint leverages the geometric nature of the polyhedral model by presenting code statements, their instances, and dependences in a scatterplot-like visualization of iteration domains similar to those used in the literature on polyhedral compilation. By making the visualization interactive, it reduces parallelism extraction and code transformation tasks to visual pattern recognition and geometrical manipulations, giving a way to manage the complexity of the underlying model.

During our preliminary design studies, interviewed domain experts expressed the following reasons for showing little interest in semi-automatic approaches: (i) the lack of immediate feedback on the effect of each transformation, (ii) the tight coupling between the code structure and the transformation specification language that complicates composition since each subsequent transformation potentially changes the code structure. Clint addresses these concerns by maintaining the consistency of the visualization independently of the underlying program structure, and by postponing the final code generation, making it possible to assess the results after each step of the transformation. For example, in Fig. 6, both blue and green statements are duplicated after transformation. Therefore subsequent transformations of the “original blue” statement require two operations with a semi-automatic tool while it can still be manipulated as one with Clint.

Finally, Clint introduces a navigable history view of transformations to favor try-and-fail strategy in the search for the best optimization. It can also save and reuse transformation scripts in order to share best practices in parallelizing optimizations. Overall, we believe that this approach may enable finer reasoning in terms of statements and instances rather than in terms of loops.

Visualizing Iteration Domains of Program Statements – Clint’s visualization of a program statement is a two-dimensional projection of its iteration domain as a polyhedron. The statement $S$ of the polynomial multiply example of Fig. 2a

---

1http://gcc.gnu.org/wiki/Graphite  
2http://polly.llvm.org
The program structure can be changed manually. The three views are synchronized and updated automatically to reflect actual changes. The source code view can however be "locked" to the original version as the transformed code tends to quickly become complex and unreadable. The history is represented as a transformation script (using Clay [3] language) with coloring and grouping to reflect visual manipulations that resulted in multiple transformations.

**Architecture and Implementation** – Clint's implementation relies on a collection of research tools and libraries that are the building blocks for polyhedral compilers and that usually work together as a single black-box [3]. Clint is a standalone application that consolidates editing facilities, feedback from the tools and output into a single consistent interface. It takes the original source code and uses Clan to raise it to the polyhedral model, then it maps direct manipulations to Clay transformation scripts and represents dependence violation feedback from Candl. Finally, it relies on ClOOG to generate the C code with OpenMP pragmas for parallelism (Fig. 8).
Finally, despite being widely used in the literature, the visualization of programs used in Clint was never evaluated. We conducted a preliminary study whose results suggest that users are able to reliably reconstruct code structure from the visualization and vice versa, which in turn suggests that the visualization is a complete representation of a program. We also asked some potential users of Clint to parallelize codes of different complexities, which they did faster and with greater accuracy than with usual methods (code editing and semi-automatic tools). While these results should be considered as preliminary, our interactive visual approach is a promising solution to ease the construction and adjustment of complex parallel optimizations of imperative programs.

REFERENCES