RenPar'18 / SympA'2008 / CFSE'6
Fribourg, Suisse, du 11 au 13 février 2008

Coupling Loop Transformations and High-Level Synthesis

Alexandru Plesco
LIP - ENS Lyon
69364, Lyon cedex 7, France
alexandru.plesco@ens-lyon.fr

Tanguy Risset
CITI - INSA Lyon
Bat 502, 20 avenue Albert Einstein
F-69621, Villeurbanne, France
Tanguy.Risset@insa-lyon.fr

Résumé

In this paper we present our study of adding an advanced preprocessing code transformation step to high-level synthesis (HLS) tools. Our approach is to use advanced state-of-the-art compiler frontend as an independent C-to-C preprocessing step before synthesis. By using this approach, recent state-of-the-art compiler advances could be used directly in HLS, eliminating their reengineering into modern HLS tools and the preprocessing effort can be reused by multiple HLS tools. We focus on efficient synthesis of loop nests and therefore we use WRaPit loop transformation framework integrated in Open64 compiler. As HLS backend we rely on Spark framework. Important improvements are obtained in the resulting RTL design thanks to the fact that WRaPit uses a polyhedral representation for nested loops and provides a flexible framework for loop transformations. Improvements are shown in particular on the synthesis of a part of the H263 decoder from MediaBench II benchmarks.

Mots-clés : outils EDA, HLS, compilation, pré-traitement

1. Introduction

Recent trends in embedded system design are platform-based design, network on chip, and higher level of specification formalisms. Intellectual property (IP) re-use is now widely considered as the main way of improving design efficiency. Embedded software design takes now a major part of a system on chip (SoC) design time. However, for widely sold products such as telecommunication devices, the use of dedicated hardware accelerators is still mandatory because it provides better trade-offs between performances (especially in terms of power consumption) and cost (chip area).

For many years, high-level synthesis (HLS) has been foreseen as the solution to accelerate dedicated hardware design. Ideally, HLS should enable the automatic generation of efficient hardware designs from functional specifications expressed in some high-level programming language. We think that HLS failed, up to now, to integrate industrial design flow because it was not mature enough to solve important technical problems :

– A huge design space to explore : potential parallelism and variety of target architecture technology imply the use of multi-criteria optimization. Some choices must be made by the designer to reduce the design space.
– The memory bottleneck : memory size and memory traffic has become a major component in chip power consumption. Optimizing memory usage is even more difficult than optimizing parallelism exploitation.
– Efficient synthesis of loop nests : while some high performance compilers have very efficient techniques to compile loop nests down to assembly code, most HLS tools implement only a subset of them.

We focus on this last problem. We show the usefulness of an independent loop transformation framework, such as the ones available in modern compilers, as a front-end to a HLS tool.

Today, existing commercial HLS tools require the original functional code (usually in C-like syntax) to be written in a very specific manner in order to get good synthesis results. Hence, a source-to-source preprocessing step is mandatory to get the code from the designer specification to a specification suitable to a particular HLS tool. Another important remark concerns the internal representation of loop nests. After many years of research in automatic parallelization, a modeling technique was developed for loop nests : the so-called *polyhedral model* [12, 17]. It provides an intermediate representation suitable for loop transformations.

Our study focuses on the combination of two tools : the Spark HLS framework [14] and the WRaPIT loop transformation tool [5], itself integrated in Open64 compiler. However, our study goes beyond these two particular tools and demonstrates that HLS can be coupled with software compilation to achieve even better results than HLS tools alone can obtain (even those that have already integrated some loop transformations like Spark, PICO-NPA and other) and in mean time to eliminate the need of compiler transformations integration internally in the HLS tool.
Section 2 briefly presents a review of HLS frameworks and recent compiler advances. Section 3 introduces our synthesis flow, which uses Spark and WRaPIT. In Section 4, we present two synthesis examples and analyze the performance improvements obtained using various loop transformations before synthesis. We conclude in Section 5.

2. Related work

There are many tools that synthesize hardware from different languages with different abstraction levels. In recent years, there is a trend in HLS frameworks to use languages with higher levels of abstraction. The higher is the abstraction, the lower are the requirements to the designer to get involved into the hardware design of the system. However, designers are often disappointed by the resulting designs, which do not correspond to what they would expect.
Considering commercial tools, after Synopsys *behavioral compiler*, there is a strong move to use C or C++ as input language. Many tools are using dedicated languages based on C-like syntax (Handel-C, Bach C, HardwareC, SpecC, etc.), introducing strong guidelines in the syntax to drive the tool towards the explicit description of the hardware. These languages are closer to hardware description languages. The major commercial tools performing HLS are CatapultC (Mentor Graphics), Pico [24] (Synfora), Cynthesizer (Forte Design System), Cascade (Critical Blue), C2H (Altera) [1]. The designer must be very familiar with these tools to get really efficient designs. Most of the time, each of these tools is efficient at synthesizing a particular type of algorithm : Pico for instance focuses on implementing efficiently perfect loop nests, possibly pipelined.
Besides, many academic tools have emerged, most of them also focused on the efficient synthesis of application-specific algorithms. Among these, the most important initiatives are : Spark [14], Compaan/Laura [27], Espam [21], MMAlpha [19], Gaut [25] , Ugh [3], Streamroller [16], xPilot [7]. Compaan, Espam and MMAlpha have focused on the efficient compilation of loops, both using the polyhedral model to perform loop analysis and/or transformations.
Compiler research is clearly imposing to be used in HLS in order to get better synthesis results. Compiler technology has led to complex re-targetable compiler frameworks such as the Gnu Compiler Collection (GCC) or the Open Research Compiler (ORC). These compilers integrate many complex optimizations, usually tuned to get efficient assembly code, but which could also be used for HLS. Examples of such optimizations are common sub-expression elimination, dead code elimination, strength reduction, to quote but a few [8, 20]. We are interested in a particular set of optimization : loop transformations [4]. Many of these transformations are already implemented in recent state of the art HLS tools like PICO-NPA, C2H and other. However, the transformations are performed mostly at a lower level of abstraction and on complex internal data structures of these tools. Our goal is to facilitate the design cycle by adding a higher level preprocessing step (human readable) that can be very easily controlled by the designer. The preprocessed specification can be synthesized by most HLS tools that take as input the specification of the system in ANSI-C.
Loop transformations were first implemented in parallelizing prototypes such as tiny [30], LooPo [13], Suif [2] or Pips [15] as they were mandatory to provide efficient parallelization. Recently, dedicated loop optimization modules have been integrated into more popular open source compilers [22], mainly because cache performances can be greatly improved by these transformations. Our goal is to use such tools to provide a source-to-source front-end to HLS tools so as to widen the space of possible hardware implementations given a particular initial sequential specification and thus eliminating the need to reimplement all these transformations internally in the HLS tools.
For applying loop transformations, we use the polyhedral model, a modeling technique for loop nests in programs. It abstracts a n-dimensional loop nest by a polyhedron on a n-dimensional space enclosing all the integer vectors spanned by the vector of indices of the loop nest. It uses a classical internal re-

presentation for the instructions of the body of the loop nest. Performing loop transformations amounts to performing algebraic transformations on these polyhedra. Recently this polyhedral representation has been successfully used to model other objects such as the memory layout of a program [9, 6], the communication volume between the IPs of a SoC [28], cache misses [18], etc.

The polyhedral representation has the advantage that its size is independent of the number of iterations of the loop. It can also manipulate loops with a parameterized number of iterations (i.e., where the number of iterations is not known at compile time). It has been used in research prototypes such as MMAlpha, Compaan or LooPo. However, its use implies a shift towards an internal representation (IR) quite different from the IR commonly used in compilers (abstract syntax trees or linear IR). Another restriction is that the polyhedral model is efficient to model *static control programs*, i.e., programs where the control flow is not dependent of the input data. As shown in [29], a major part of programs is composed of static control parts, especially for computationally intensive programs present in signal processing or multimedia applications. However, it is mandatory for a HLS tool to also handle the parts of the program that do not have static control.

We have chosen the `WRaPIT` tool [5] because it explicitly implements a polyhedral internal representation, and it is integrated in the `Open64` compiler. This allows us to rely on `Open64` for the non-static control parts of programs. Another important point is that `WRaPIT` is not a parallelization tool, it manipulates sequential code. This is important because we think that parallelization should be handled by the HLS tool as it is very dependent on the target design technology. Another useful property of `WRaPIT` is its user interface for loop manipulation. The user can very easily specify loop transformations and can provide new ones thanks to the `uruk` script language.

The work presented here could also have been done with a dedicated source-to-source tool such Nestor [26] or Rose [23] instead of a complete compiler. We could also have used parallelizing tools such as LooPo or Tiny. The important properties that the loop transformation tool must have are : `i`) it must contain a complete loop transformation module to handle parameterized loops, `ii`) it must be able to input and output the given source language from which the synthesis starts, in our case, the C language.

Independently of our work, an interesting study has been recently published [10] that uses `WRaPIT` as loop transformation tool too. The HLS was done with an elementary (i.e., no complex back-end optimizations) homemade synthesis tool (`CloogVHDL`) and its results were compared to those obtained by ImpulseC, starting from the same initial specification. We have a different goal. There are many HLS tools that have already included some compiler technologies and that have already proved their efficiency. However, most of them implement a small subset of compiler transformations. Implementing a new compiler transformation requires very good knowledge of the internals of these tools. Another problem that we are trying to solve is the possibility of adding a new transformation to an existing HLS tool. We pointed out earlier that it is not very easy to add one to a HLS tool when the source code is available, in the mean time it is impossible to add one to a HLS tool provided in binary format (like `Spark`, `PICO-NPA` and most of the proprietary HLS tools). Our design flow allows an easy, human readable code preprocessing step to most of them.

As a backend, we prefer to rely on a HLS tool such as `Spark` for the following reasons. First, it is important to take into account the possible interactions of high-level transformations with back-end optimizations. Possible performance loss cannot be clearly seen with a basic HLS tool, so `Spark` was a better choice. Second, we chose it because its main strength are control-intensive programs and thus we can rely on the optimized FSM output since loop transformations will alter the complexity of the generated FSM. Third, relying on independent tools for both parts (front-end and back-end) shows the feasibility of our two-phases approach. Indeed, the fact that we use `Spark` as a black box, with no possible source modification, shows that we could do the same with a commercial tool. Finally, using sophisticated tools for the front-end and back-end brings the best of the two worlds. For example, we can consider a full application, not just static-control programs, even if we perform loop transformations only on static-control parts.

We point out that, unlike [10], our loop transformations are selected manually so far. However, we can analyze the impact of every single transformation (or a series of transformations) on the resulting hardware (size, latency, etc.).

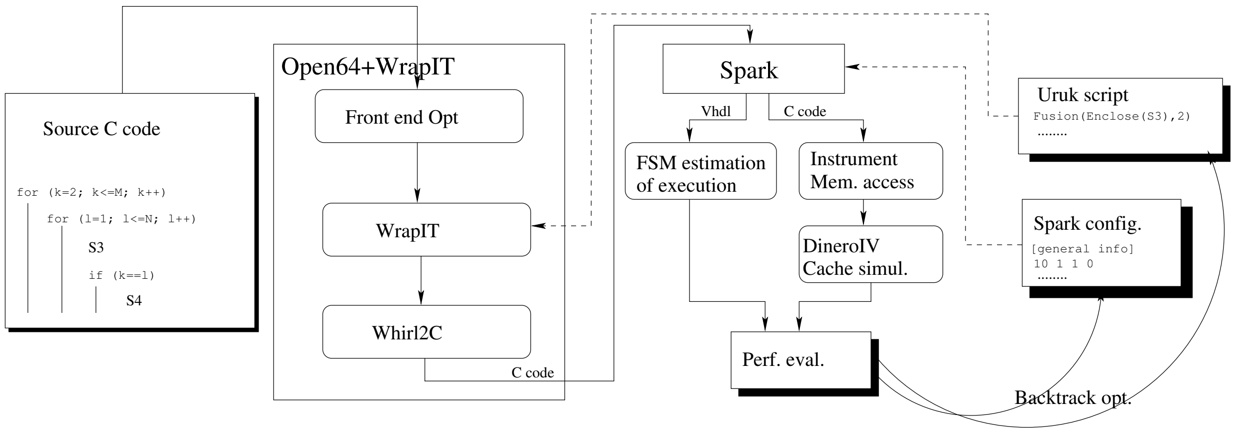


FIG. 1 – Our VHDL design flow combining WRaPit and Spark.

3. Design flow with Spark and WRaPIT

In our design flow (Fig. 1), we used the `WRaPIT` framework [5], the `Open64` compiler, the `Spark` HLS framework [14], and the `dineroIV` cache simulator [11].
`WRaPIT` is currently plugged in the `Open64` compiler and replaces its loop nests optimizer with a more powerful one, as proved by benchmarks [5]. `Spark` is an academic HLS tool, provided in binary format, that takes as input a subset of ANSI-C code and incorporates many code transformation techniques to improve the quality of the synthesized circuit. `Spark` generates a synthesizable VHDL state machine, produces performance statistics, and is able to generate C code that emulates the parallel execution of the hardware generated. However, inputs and outputs are supposed to be wired, and no memory controller is provided.

3.1. Target architecture

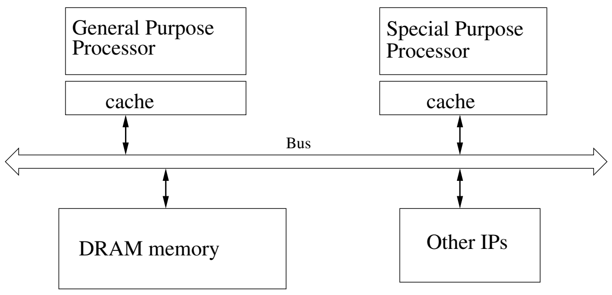


FIG. 2 – Target architecture model

In order to compare HLS performances of different designs, a target system architecture class must be described. Most high-level synthesis frameworks generate architectures for a co-processor that will run together aside to a general-purpose processor. The target architecture we used is represented in Fig. 2, our special-purpose processor synthesized by `Spark` is connected to the bus via a cache. Our work focuses on the design of the special-purpose processor and its interaction with the external memory. The cache memory may be external or integrated into the circuit.
We did not synthesize any memory controller but we simulated the effect, on the cache, of the IP generated by `Spark`, thanks to `dineroIV`, an open-source tool for cache trace-driven simulation of various

cache configurations. dineroIV was configured to a size of 8KB with a block size of 32B and associativity 4 for the first example and 32KB with a block size of 64B and associativity 8 for the second one and for both with a LRU replacement policy, fetch demand policy, write allocation always, and write back always. The write allocation policy was chosen because of the spatial locality of writes found in multimedia applications as well as in our example. This improves dramatically the burst write modes to the external memory. Synthesis results are obtained by instructing Spark with the following timing constraints and resource constraints : 20ns clock cycle and 10 of each of the arithmetic operators available (ALU 10ns, MUL 20ns, ...).

3.2. Design flow

We start from an initial sequential C specification (Fig. 1). This code is fed to Open64. In the front-end of the Open64 multiple code transformations are performed such as procedure inlining, dead function/variable elimination, constant propagation, etc. The use of procedure inlining is to transform a code having multiple function calls to a code that is ready for synthesis (i.e., without function calls). In the next step, loop transformations are performed by WRaPIT, guided by the user (the user indicates which transformation to perform thanks to the uruk script language). The Whirl2C program, provided with Open64, is used to generate C code. At this step, because of some bugs found in Whirl2C at the generation, we needed to modify this code. Then, this modified C program is fed to Spark, which is configured with resource constraints mentioned in Section 3.1. Spark outputs a VHDL file describing the circuit and a C file describing the implementation. The performances of the resulting circuit are evaluated from these files. We instrumented the C file produced by Spark. This C file, when compiled and executed, generates a trace of memory accesses, which is finally used by the cache simulator to analyze memory access performances.
We point out that the VHDL designs generated by Spark cannot be synthesized directly. As previously mentioned, Spark does not synthesize a memory controller for its inputs and outputs, instead the circuit contains many I/O pins. Each I/O data bit generates a new I/O pin. Since it was conceived for control-intensive programs it does not generate a real memory controller (we point out that we have chosen it because of its state of the art internal optimizations). However, the schedule of the I/O operations made by Spark is correct and the hardware generated for the computational kernel is valid, i.e., it respects resource constraints. Nevertheless, our evaluation with the cache simulator is realistic.

4. Experiments

In this section, we first present an ad-hoc example to highlight the potential gains that loop transformations can bring to HLS designs, then we provide other performance results on a real H263 decoder application.

4.1. An illustrative example

The initial example (Fig. 3) consists of a C code performing edge detection on a 100 by 100 pixels 16 bit depth image : it first applies the Laplacian filter to an image, then applies horizontal and vertical Sobel filters on the image. This code contains many temporary arrays, it is typically written by an application designer who does not take into account memory access optimizations but would rather like to have a readable code to tune up the algorithm. This code is synthesized by Spark. The cache size setup for this example is 8KB with a block size of 32B.
The loop transformations applied to the example using WRaPIT are classical ones : code motion in order to move the initialization of arrays next to their use, a sequence of loop fusion and temporary array elimination. This code, once synthesized by Spark, is likely to have much better performances because simultaneously i) many of the intermediate arrays are removed, hence memory traffic is reduced, ii) the number of loops is reduced, reducing the control synthesized for each loop by Spark.
The results obtained on this example are presented in Fig. 4 for three versions of the source C-code : the initial C program, the initial program with an unrolling by 20 of the innermost loop (performed by Spark), and the initial program optimized with WRaPIT. We point out that at the end of every experiment we didn't performed a cache flush, thus if required, the real memory bandwidth size can be obtained by adding the cache size and the size of the write buffer. This figure presents the improvements of the resulting hardware designs in terms of number of cache misses, total number of communications

```
for (i = 1; i < N-1; i++)
  for (j = 1; j < N-1; j++)
  L3:  A1[i*N+j] = (-1) * A[(i-1)*N+j-1] +
      (-1)*A[(i-1)*N+j] + (-1)*A[(i-1)*N+j+1] +
      (-1)*A[i*N+j-1] + (8)*A[i*N+j] +
      (-1)*A[i*N+j+1] + (-1)*A[(i+1)*N+j-1] +
      (-1)*A[(i+1)*N+j] + (-1)*A[(i+1)*N+j+1];
// Horizontal Sobel filter stores the
// result in B1 array
for (i = 1; i < N-1; i++)
  for (j = 1; j < N-1; j++)
  L4:  B1[i*N+j] = (-1) * A[(i-1)*N+j-1] +
      (2)*A[(i-1)*N+j] + (-1)*A[(i-1)*N+j+1] +
      (-1)*A[(i+1)*N+j-1] + (2)*A[(i+1)*N+j] +
      (-1)*A[(i+1)*N+j+1];
// Vertical Sobel filter stores the
// result in B2 array
for (i = 1; i < N-1; i++)
  for (j = 1; j < N-1; j++)
  L5: B2[i*N+j] = (-1) * A[(i-1)*N+j-1] +
      (-1)*A[(i-1)*N+j+1] + (2)*A[i*N+j-1] +
      (2)*A[i*N+j+1] + (-1)*A[(i+1)*N+j-1] +
      (-1)*A[(i+1)*N+j+1];
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
  L6: C[i*N+j] = A1[i*N+j] + B1[i*N+j] +
      B2[i*N+j];
```

(a) initial code

```
motion_block(LBL2,LBL6)
fusion(enclose(LBL4,2))
fusion(enclose(LBL4))
fusion(enclose(LBL3,2))
fusion(enclose(LBL3))
fusion(enclose(LBL5,2))
fusion(enclose(LBL5))
fusion(enclose(LBL1,2))
fusion(enclose(LBL1))
```

(b) Uruk script

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
  L6: if ((i == 0)||(i == N-1)||
         (j == 0)||(j== N-1))
         C[i*N + j] = A[i*N + j];
    else
    C[i*N + j] = (-3) * A[(i-1)*N + j-1] +
    (-3)*A[(i-1)*N+j]+(-1)*A[(i-1)*N+j+1] +
    (-3)*A[i*N+j-1]+(8)*A[i*N+j] +
    (1)*A[i*N+j+1]+(-1)*A[(i+1)*N+j-1] +
    (1)*A[(i+1)*N+j]+(1)*A[(i+1)*N+j+1];
```

(c) final code

FIG. 3 – Source code before and after loop transformations with WrapIT

between the accelerator and the memory, and total number of execution clock cycles. The *ideal* number of cycles is the number of cycles evaluated by Spark, which assumes no cache miss (i.e., 1 cycle response for any data access), the *simulated* number of cycles is computed assuming a 40 cycles latency for each cache miss. With unrolling, Spark can perform parallelization and loop pipelining more efficiently. This explains the improvement in the number of clock cycles. However, the transformations performed with WRaPIT (which does not include any unroll) still has better performance on each metric. This experiment presents dramatic improvements : 5.57X speedup in the number of total cycles, 6.97 times better in the number of cache misses. An analysis of the final code (Fig. 3(c)) for the reuse distances of the elements of the array A can show that after the optimizations the synthesized architecture does not need a cache at all but only a small internal buffer.

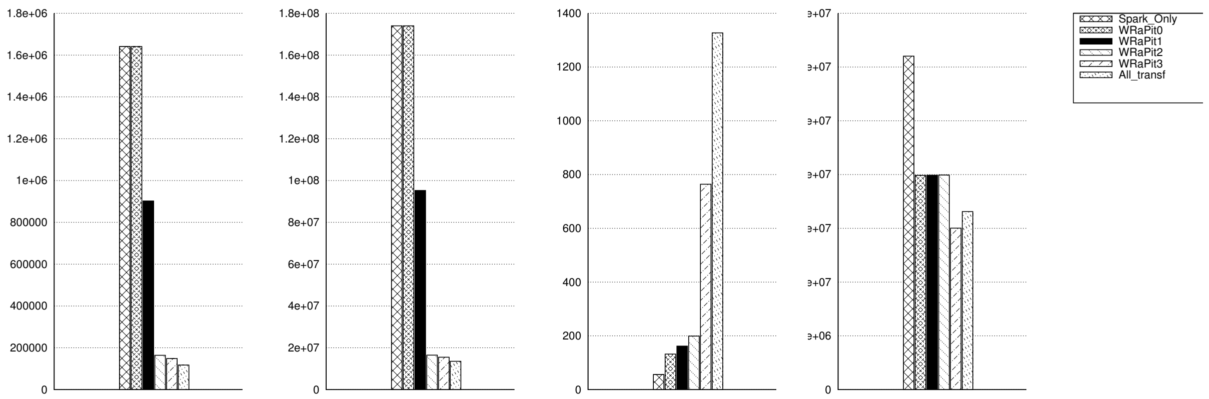
	cache miss	mem. (#bytes)	#cycles ideal	#cycles real
Spark alone	4 364	219 200	273 077	447 637
Spark unroll j 20	4 364	219 200	74 969	249 256
Spark+WRaPIT	626	30 048	55 302	80 342

FIG. 4 – Performance improvements on edge detection algorithm (cache size of 8 KB, bsize of 32)

4.2. H263 decoder

We now present the same performance improvements on a more realistic example taken from Media-Bench II Benchmark[1] : a H263 decoder. Profiling of the decoder shows that an important part of the execution time (62.33%) is taken by the YUV to RGB conversion (vertical interpolation, horizontal interpolation and the YUV to RGB conversion). This color space conversion is present also on all the video decoding codecs like mpeg4 and image decoding ones as jpeg. On a desktop computer, this conversion

[1] http://euler.slu.edu/~fritts/mediabench/mb2/



(a) Cache Miss (b) Memory R/W (c) FSM States (d) Execution cycles

FIG. 5 – Performance results of the synthesis

is done by the video card. Most of the embedded systems do not have one and this conversion is done by the CPU, a VLIW processor, or a hardware accelerator. Inlining was used as a preprocessing step to get all computations in a single function. The function performs, in a sequence of doubly-nested loops, a series of transformations on two arrays : U and V.

The synthesis and simulation were done on a fixed 1000x1000 pixels image. The synthesis results presented in Fig. 5 are obtained by applying various loop transformations as summarized below :

- Spark_Only : no use of WRaPIT.
- WRaPIT0 : Passing through WRaPIT without loop transformations.
- WRaPIT1 : loop reversal on vertical interpolation (U).
- WRaPIT2 : WRaPIT1 plus loop reversal on vertical interpolation (V).
- WRaPIT3 : WRaPIT2 plus loop transformations (fusion, strip-mining and shifting) on horizontal and vertical interpolations of U and V.
- All_transf : WRaPIT3 plus loop transformations (code motion, strip-mining, loop interchange) on U and V and RGB loops and multiple loop fusions.

4.3. Results and discussions

We now present the performance results obtained from the Spark synthesis report (number of clock cycles) and from the dineroIV cache simulator using the C code output by Spark. Fig. 5 illustrates the results for cache behavior (cache misses and effective R/W in memory), total number of clock cycles, and the FSM size of the generated hardware.

There is an improvement in the number of cycles between Spark_only and WRaPIT0 because, in the original code, a lot of if statements are used to check border conditions. This can be simplified by CLooG (the loop generator of WRaPIT). The WRaPIT1 transformation performs a reversal of the two loops of the vertical interpolation on U. Because the original interpolation is done vertically and the array elements are stored horizontally, the reversal improves cache miss ratio thanks to spatial locality. The WRaPIT2 transformation brings similar improvements.

The WRaPIT3 transformation improves the temporal locality between the vertical interpolation and the horizontal one. Because of the data dependencies between them, a shift by 2 operation was performed on a loop of the vertical interpolation. An important improvement of the number of execution cycles can be observed after this transformation because most of loop control hardware is shared between the two fused loops. There is also a sharing of the array access index calculations that are in the critical path. This transformation is also used to increase the level of the parallelism inside the loops that Spark can easily explore. However, this can also perturb smooth cache operations and thus may increase the

cache miss ratio. The last transformations (All_transf) improves the temporal locality of the writes for the previously-obtained nested loops. The performance results of the cache misses are improved even more. As it can be observed from Fig. 5(d), the number of execution cycles increases as compared to WRaPIT3. In this case, the performance degradation of the strip-mine transformation can be very clearly observed. Each strip-mine transformation is generating a new loop nested inside the original loop nest. The cache locality improvement brought by the strip-mine transformation has to be balanced with the improvement of the number of cycles.
As previously mentioned, the hardware generated by Spark can not be synthesized directly, however one can have a rough idea of its area complexity : the hardware consists of an execution unit, the same for each design, and a finite state machine controlling the execution unit. Fig. 5(c) gives the number of states of the FSM. It increases for each new transformation, especially for WRaPIT3 and All_transf. Here again, there is a tradeoff between cache performance and hardware complexity.
The goal of these experiments was to show that important improvements can be obtained by using loop transformations as a front-end to HLS, especially if these loop transformations are guided by the user (automatic loop optimization is still not applicable to HLS). Similar improvements were obtained in [10]. The main benefit of our approach is that it provides the flexibility of adding a powerful loop transformation front-end to existing HLS frameworks, even if the HLS framework is provided in binary format. Of course, this applies only if the synthesis framework accepts ANSI-C as input.

5. Conclusions and future works

In this paper, we show that an independent compiler optimization step should be used as front-end to HLS tools. There are already many HLS tools that perform some compiler optimizations, however the implementation of additional loop transformation steps is not an easy task : it requires an additional engineering effort and in some cases it is even impossible (for binary available ones). The novelty of our approach is that it demonstrates the feasibility of coupling an independent preprocessing step by using a modern state-of-the-art compiler performing loop transformations to a HLS tool. This approach facilitate the use of recent state-of-the art compiler advances in HLS, eliminates the reengineering effort and enables the reuse of preprocessing effort by multiple HLS tools. We demonstrate the potential gains of this approach by using the Open64 compiler, coupled with the WRaPIT tool, as a front-end to the Spark HLS tool that can also perform internally some basic loop transformations. As a result, additional loop transformations could be applied before synthesis increasing dramatically the performances. Since the result of the preprocessing step is human readable (C code) it can be analyzed in a very short time (compared to the time of actual hardware synthesis). This work was based on academic tools but similar combinations are possible with most of the HLS tools accepting ANSI-C as input. Indeed, HLS is currently more and more used in industrial systems for chip design and WRaPIT is currently being integrated in GCC, with the name Graphite [22].
Preparing the code for HLS tools, with high-level transformations, is not new, but it is done by hand. Indeed, users of commercial tools have to perform this tedious task by hand, not necessarily for performance but also just to make the C code syntactically acceptable for the tool. Our study shows that, instead of doing it by hand, it can be done efficiently, faster and error free in a semi-automatic way by selecting high-level transformations provided by a front-end compiler. The next step of this research is to build a complete toolbox that will include all useful transformations for HLS back-end tools.

Bibliographie

1. Automated Generation of Hardware Accelerators With Direct Memory Access From ANSI / ISO Standard C Functions, 2006. Altera.
2. S. Amarasinghe et al. Suif : An infrastructure for research on parallelizing and optimizing compilers. Technical report, Stanford University, May 1994.
3. I. Augé, F. Pétrot, F. Donnet, and P. Gomez. Platform-based design from parallel C specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(12) :1811–1826, 2005.
4. David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4) :345–420, 1994.

5. Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. Putting polyhedral loop transformations to work. In *LCPC*, pages 209–225, 2003.
6. F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology*. Kluwer Academic Publishers, 1998.
7. J. Cong, Yiping Fan, Guoling Han, Wei Jiang, and Zhiru Zhang. Platform-based behavior-level and system-level synthesis. In *International SOC Conference*, pages 199–202. IEEE, 2006.
8. Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan-Kaufmann, 2003.
9. Alain Darte, Robert Schreiber, and Gilles Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10) :1242–1257, 2005.
10. Harald Devos, Kristof Beyls, Mark Christiaens, Jan Van Campenhout, Erik H. D'Hollander, and Dirk Stroobandt. Finding and applying loop transformations for generating optimized fpga implementations. *Transactions on High Performance Embedded Architectures and Compilers I*, 4050 :159–178, 2007.
11. Dinero IV Trace-Driven Uniproc. Cache Simulator. `http://www.cs.wisc.edu/~markhill/DineroIV/`.
12. Paul Feautrier. Automatic parallelization in the polytope model. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, volume 1132 of *LNCS Tutorial*, pages 79–103. Springer Verlag, 1996.
13. M. Griebl and C. Lengauer. The loop parallelizer LooPo. In Michael Gerndt, editor, *Proc. 6th Workshop on Compilers for Parallel Computers*, volume 21 of *Konferenzen des Forschungszentrums Jülich*, pages 311–320. 1996.
14. S. Gupta, R. Gupta, N. Dutt, and A. Nicolau. *SPARK : A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Kluwer Academic, 2004.
15. F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization : An overview of the PIPS project. In *ACM Intern. Conf. on Supercomputing, ICS'91*, 1991.
16. Manjunath Kudlur, Kevin Fan, and Scott Mahlke. Streamroller : Automatic synthesis of prescribed throughput accelerator pipelines. In *CODES+ISSS'06 : Proc. of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, pages 270–275, New York, NY, USA, 2006. ACM Press.
17. C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.
18. Vincent Loechner, Benoît Meister, and Philippe Clauss. Precise data locality optimization of nested loops. *The Journal of Supercomputing*, 21(1) :37–76, 2002.
19. A. Mozipo, D. Massicotte, P. Quinton, and T. Risset. Automatic synthesis of a parallel architecture for Kalman filtering using MMAlpha. In *International Conference on Parallel Computing in Electrical Engineering (PARELEC 98)*, pages 201–206, Bialystok, Poland, September 1998.
20. Steven S. Muchnick. *Compiler Design Implementation*. Morgan-Kaufmann, 1997.
21. Hristo Nikolov, Todor Stefanov, and Ed Deprettere. Efficient automated synthesis, programming, and implementation of multi-processor platforms on FPGA chips. In *16th Int. Conference on Field Programmable Logic and Applications (FPL'06)*, pages 323–328, Madrid, Spain, August 2006.
22. S. Pop, G.-A. Silber, A. Cohen, C. Bastoul, S. Girbal, and N. Vasilache. GRAPHITE : Polyhedral analyses and optimizations for GCC. Technical Report A/378/CRI, CRI, ENSMP, Fontainebleau, France, 2006.
23. Daniel J. Quinlan. ROSE : Compiler support for object-oriented frameworks. *Parallel Proc. Letters*, 10(2/3) :215–226, 2000.
24. Robert Schreiber, Shail Aditya, B. Ramakrishna Rau, Vinod Kathail, Scott Mahlke, Santosh Abraham, and Greg Snider. High-level synthesis of nonprogrammable hardware accelerators. In *ASAP'00*, page 113, Washington, DC, USA, 2000. IEEE Computer Society.
25. O. Sentieys, J.P. Diguet, and J.L. Philippe. Gaut : A high level synthesis tool dedicated to real time signal processing application. In *European Design Automation Conference*, September 2000. University booth stand.
26. Georges-André Silber and Alain Darte. The NESTOR library : A tool for implementing FORTRAN source transformations. In *High-Performance Computing and Networking Europe*, pages 653–662, 1999.
27. Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. System design using Kahn process networks : The Compaan/Laura approach. In *DATE'04 : Proc. of Design,*

Automation and Test in Europe, pages 340–345, Washington, DC, USA, 2004. IEEE Computer Society.

28. Alexandru Turjan, Bart Kienhuis, and Ed F. Deprettere. Translating affine nested-loop programs to process networks. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 220–229, 2004.
29. Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. Polyhedral code generation in the real world. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'06)*, LNCS, pages 185–201, Vienna, Austria, March 2006. Springer-Verlag.
30. M. Wolfe. A loop restructuring research tool. Technical Report CSE 90-014, Oregon Graduate Institute, August 1990.