Using Rewriting Logic to Match Patterns of Instructions from a Compiler Intermediate Form to Coarse-Grained Processing Elements

Carlos Morra[1] João M. P. Cardoso[2] Jürgen Becker[1]

[1] Institut für Technik der Informationsverarbeitung (ITIV)
Universität Karlsruhe (TH), Karlsruhe, Germany
{morra,becker}@itiv.uni-karlsruhe.de

[2] UTL/IST, Department of Informatics Engineering
INESC-ID, 1000-029, Lisbon, Portugal
jmpc@acm.org

Abstract

This paper presents a new and retargetable method to identify patterns of instructions with direct support in coarse-grained processing elements (PEs). The method uses a three-address code SSA (static single assignment) representation of the kernel being mapped and Rewriting Logic for template matching and algebraic optimizations. This approach is able to identify sets of SSA instructions that can be mapped to different PE complexities available in coarse-grained reconfigurable computing architectures. As a proof of concept, results of the approach with a number of benchmark kernels, as far as coverage of template instructions is concerned, are included.

1. Introduction

The use of VLIW-based templates in order to accelerate loop intensive behavior has been recently focus of renewed research efforts [1]. VLIW-based approaches rely on 1-dimension (1D) of PEs (typically ALUs, multipliers, load/store units, etc.) directly used to execute typical single and two operand arithmetic and logical operations. They use a register file to store data loaded from main memory and/or computed by the PEs. More advanced VLIW templates use clusters of simple VLIW architectures (PEs and register file) in a distributed scheme. On the other hand, coarse-grained reconfigurable architectures [2] are not tied to a simplistic template (such as the VLIW one). They can use different routing topologies, 1D or 2D matrixes of PEs, PEs with support to more complex operations, heterogeneous or homogeneous PEs, distributed memories, etc. In this aspect, a VLIW template can be thought as a specific case of coarse-grained reconfigurable array architectures.

Although also suitable to identification of templates of instructions for generating ISEs (Instruction Set Extensions), the work presented in this paper bears in mind the exploration of different coarse-grained architecture templates based on 1D or 2D arrays of PEs. One of the interesting design explorations is to evaluate the impact on performance when different PEs supporting complex operations are used. However, the exploration needs a retargetable compiler able to identify instructions that resemble templates directly supported by the PE's. For such exploration, this paper shows an approach using an SSA (Static Single Assignment) form [3], composed by a three address based representation output generated by the ***Nau*** compiler [4] from the Java bytecodes of a given method [5].

Figure 1 presents the framework under development. The environment uses an extended SSA form and a Term Rewriting System (TRS) to identify patterns of instructions. Note, however, that the integration of a simulator engine and techniques to estimate performance results with different coarse-grained architectures are currently the focuses of our work and are not the scope of this paper.

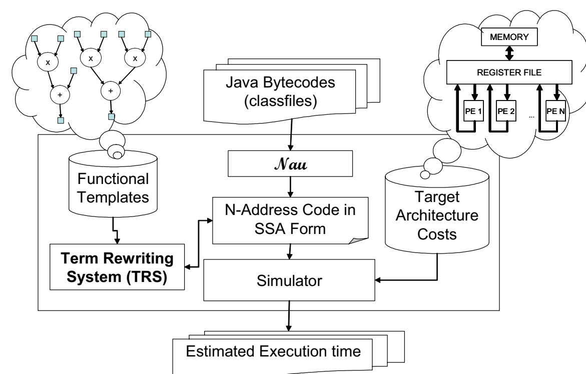


Figure 1. Proposed exploration environment.

The main contributions of this paper are:

- The approach presented herein is able to perform identification of instruction templates bearing in mind different goals: generation of Instruction Set Extensions and mapping groups of instructions to PEs;
- Template matching is performed across basic block boundaries, which can be important when PEs implement structures using instructions disperse in the SSA form (e.g., counters to implement loop control behavior);
- It is shown how this can be exploited using Rewriting Logic and an extended SSA intermediate form from a compiler;
- It illustrates the advantages the rewriting-logic paradigm provides in this context;

1-4244-0910-1/07/$20.00 ©2007 IEEE

- Additionally, the approach is able to consider optimizations (e.g., expression tree transformations and algebraic simplifications) to achieve better template matching results;
- To the best of our knowledge this is the first time Rewriting Logic is used to accomplish the referred goals;
- Experimental results to validate the concept are presented for different instruction templates applied to a number of kernels from image and signal processing domains.

This paper is structured as follows. Next section introduces coarse-grained reconfigurable architectures. Section 3 explains concepts about term rewriting and rewriting logic. In Section 4 the intermediate representation is presented. The proposed methodology is explained in Section 5, and in Section 6 experimental results are presented. In Section 7, the related work is introduced and discussed. Finally, Section 8 draws some conclusions.

2. Coarse-Grained Array Architectures

Coarse-grained array architectures [2] consist of a number of PEs interconnected by certain routing topologies. Various architectures have been proposed with different routing topologies and/or different types of functional units (FUs) that can be implemented by each PE (e.g., Morphosys [6], ADRES [7], PACT XPP [8]).

Regarding each PE's functionality, the simplest ones consider multiplier and ALU operations on each PE. Each PE usually has two inputs and one output. In this case, simple one or two-operand arithmetic operations can be directly mapped to each PE.

There are architectures that use more complex PEs. As an example, the architecture proposed in [9] uses a coarse-grain component for each PE. In this kind of architecture, a large number of groups of operations can be implemented using each PE. For instance, we may program a PE to execute A×B+C×D, A×B-C×D, A×B+D, A+B+C+D, A+B+C, etc. Thus, the kind of patterns of operations we may be able to map to a single PE largely depends on the target architecture. Another example of PE's complexity is the support to implement a counter in a single PE of the architecture, as is the case in the XPP [8].

To explore the large design space we need a strategy able to map an input program representation to the target architecture based on the specification of the templates supported by each PE. Next section presents the foundations behind the novel strategy proposed in this paper.

3. Term Rewriting and Rewriting Logic

Term Rewriting [10] is the formal mathematical framework for the reduction of expressions using matching and substitution of terms. Term rewriting is applied in the form of rewriting rules that define how the term is transformed.

Rewriting rules are of the form:

$$s \rightarrow t \text{ if } c$$

Meaning that a sub-term that matches the left-hand side of the rule will be replaced by the right-hand side when the condition "c" holds. These operational semantics are the same as those involved in functional environments and have been promoted in functional programming languages since the well-known McCarthy LISP of the 1950s.

Rewriting-logic is the result of using logic strategies to control how and when the term-rewriting rules are applied. Thus, rewriting-logic allows for different paths of reduction and the possibility of obtaining different forms for the same input term.

The use of rewriting-logic in the context of retargetable compilers and design space exploration has many advantages:

- Expressiveness: its very simple operational semantics avoid the inclusion of all the unnecessary semantics that are required in programming languages.
- Simplicity: features like automated type checking, multiple types per variable, expression tree traversing, lexical analysis, pattern matching for both terms and sub-terms, transformation mechanisms, etc. are already built in Term Rewriting Systems. For this reason it is easier and faster to define and develop the instruction patterns using rewriting-logic than with traditional computer languages.
- Powerfulness: it provides exceptional capabilities for finding very complex patterns of instructions. It provides natural mechanisms for expression tree transformations, algebraic manipulation and modeling reconfigurable systems.

Term Rewriting Systems can be efficiently implemented using term rewriting computational environments, being two of the most popular ELAN [11] and Maude [12].

4. Intermediate Representation

The SSA form [3] with three address code format (see Figure 2 and Figure 3 for an example) is used as the starting point of our approach. As can be seen in Figure 3(a), this representation uses arithmetic operations (e.g., IADD, GE, SHR, IMUL), assignments of constants or variables to variables (ASSIGN), selection points (e.g., MICRO and PHI), load/stores operations (e.g., sLd, sSt, iLd, iSt), jump instructions (JUMP) and return instructions (RETURN). The instructions are grouped in basic blocks (BB #). Note that for simplicity, this example does not include the bit-width of each variable which is present in the format output from the compiler. The identifiers using a dot and two integers represent variables (unique name for each assignment) and

identifiers with only an integer value represent constants. Instructions such as GE (greater or equal) besides the comparison jump to the basic block represented whenever the condition evaluates to true. By default, a subsequent basic block is executed after the current one (with the exception in the presence of branches). The instructions (e.g., GE, SHR, IADD), the inputs (variables and/or constants), and the output variables are separated by '|'.

```
for (short j = 0; j < Nx; j++) {
    int sum = 0;
    for (short i = 0; i < Ntaps; i++) {
        sum += x[i + j] * h[i];
    }
    y[j] = (short) (sum >> 15);
}
```

Figure 2. FIR example.

```
BB #0
1: [ ASSIGN | 0 | 5.0 ];
BB #1
2: [ MICRO | 5.0, 5.2 | 5.1 ];
3: [ GE | 4.0, 5.1| 9.0 | BB #6 ];
BB #2
4: [ ASSIGN | 0 | 6.0 ];
5: [ ASSIGN | 0 | 7.0 ];
BB #3
6: [ MICRO | 7.0, 7.2 | 7.1 ];
7: [ MICRO | 6.0, 6.2 | 6.1 ];
8: [ GE | 3.0, 7.1| 12.0 | BB #5 ];
BB #4
9: [ IADD | 5.1, 7.1 | 15.0 ];
10: [ sLd | 15.0, 0.0 | 14.0 ];
11: [ sLd | 7.1, 1.0 | 15.1 ];
12: [ IMUL | 15.1, 14.0 | 14.1 ];
13: [ IADD | 14.1, 6.1 | 6.2 ];
14: [ IADD | 1, 7.1 | 7.2 ];
15: [ JUMP | BB #3 ];
BB #5
16: [ SHR | 15, 6.1 | 17.0 ];
17: [ sSt | 17.0, 5.1, 2.0 ];
18: [ IADD | 1, 5.1 | 5.2 ];
19: [ JUMP | BB #1 ];
BB #6
20: [ RETURN ];
```

(a)

```
[
PE3x1BA |15.1,14.0, 6.1| 6.2
  {
    [IMUL |15.1,14.0 | 14.1],
    [IADD |14.1, 6.1| 6.2]
  }
];
```

(b)

Figure 3. SSA form output by the Nau compiler for the FIR example: (a) original; (b) Instructions 12 and 13 grouped to be implemented as a PE3x1BA.

This representation is generated with the ***Nau*** compiler [4] and has been selected as the input intermediate representation for the Rewriting Logic due to the following reasons:

- It maintains a high-level of abstraction without details of the target architecture and thus it can be efficiently used both to generate assembly code for a typical microprocessor, VLIW, or to coarse-grained reconfigurable architectures;
- It represents the behavior of imperative programming languages (note that in this case the intermediate representation is obtained from the Java bytecodes);
- It can be seen as a textual representation of dataflow graphs, because each symbolic variable can be thought as a connection. This property permits to declare and apply template rules without needing control- and dataflow analysis or graph construction;
- It is an intermediate representation model that can be suited to dynamic compilation;
- It is the intermediate representation used by modern compilers to perform static analysis and many optimizations.

Each group of instructions mapped to the same PE is identified by a label representing the PE (e.g., P3×1BA in Figure 3(b)). This way, the initial specification required for generating the code to program the PEs of the target architecture, and for the simulation step is maintained. An important additional step deals with the maintenance of redundant instructions as is explained in next section.

5. Methodology

The proposed methodology is supported by the steps illustrated in Figure 4. The input of the methodology is the SSA representation of a function or procedure. The Rewriting-Logic rules and strategies are used to group sets of SSA instructions that can be directly mapped to the PEs of the target architecture. The capabilities of the PEs are defined as a set of templates expressed as Term Rewriting rules.

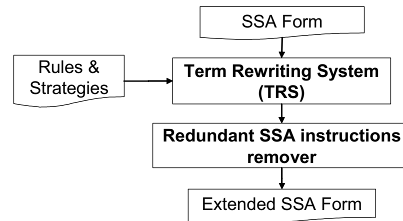


Figure 4. Steps of the approach.

It is important to note that a PE may not have outputs for all its internal intermediate blocks, therefore it may be necessary to duplicate one or more of the intermediate instructions in the cases where their results are used in other nodes of the expression tree. The current version maintains the original instructions and adds copies of the grouped instructions in the TRS step. Those replicated instructions which are not required are automatically removed in the following step (see Figure 4).

The Maude System [12] with the strategy language extensions [13] is used to implement the proposed methodology. Rewriting-Logic is used to exploit the mapping of high-

level procedural programming languages to coarse-grained reconfigurable arrays using the following optimizations:

- Mapping of groups of instructions to the expressions directly supported by the PEs of the target architecture under evaluation (e.g., merging operation trees into multiple input operators such is the case when merging a MUL-ADD tree into a MAC instruction);
- Expression tree transformations. E.g., tree height reduction to decrease the critical path delay;
- Algebraic optimizations applying transformations such as commutative, associative, etc. These transformations may increase the potential for template matching;
- Identification of counters related to, e.g., loop iteration control;
- Performing operator strength reduction (e.g., mapping of multiplications by constants to shifts and additions/subtractions);

Other optimizations that are planned to be included are: decomposition of instructions into subparts. E.g., decomposing a 32-bit operation into 16-bit operations; merge of operations to be implemented as SIMD (Single Instruction Multiple Data) operations; merge of operations working on packed data.

The grammar of the SSA intermediate form was defined in the TRS as two modules: one for the basic grammar structure and the second one for the instruction set. The types defined for the basic grammar are shown in Table I. The syntax of the grammar is described as operators in the TRS. The opcodes and the syntax for the PEs were defined in the same way, but adding a section with the functional behavior of the PE.

Table I. Types defined in the basic grammar TRS module.

Type	*Used for*
SingleExpression	A single statement (subtype of Expression)
Expression	One or more statements
VariableNumber	Variable identifiers
Constant	Constant identifiers
Variable	Both variable and constant identifiers with the number of bits
Opcode0	Instructions without parameters
Opcode[1-5]	Instructions with 1 to 5 operands
OpcodeB	Block labels
OpcodeC	Conditional instructions
OpcodeJ	Jump instructions
OpcodeR	Return instructions

Each template of the operations that can be mapped to each PE (some of the exploited PEs can be seen in Figure 5) was written as a set of rewriting rules. The list of some implemented rules is presented in Table II. Table III shows some rule definitions related to the template PE3×1BA.

The application of the term rewriting rules is controlled by logic strategies. The simplest strategy is to select one set of rules and apply them until the term is in a normal form (i.e., the term cannot be further reduced by the selected rules). A slightly more complex strategy is to normalize the term by using a sequence of different rules. These two types of strategies were used to analyze the maximum coverage of each template for each benchmark. A list of the implemented strategies and their sequence of rules is given in Table IV.

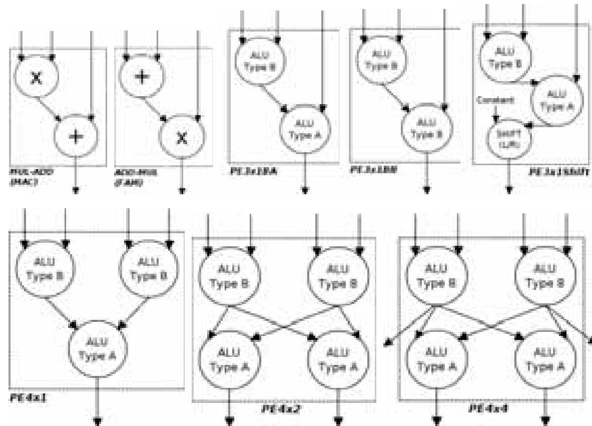


Figure 5. Examples of instruction templates being exploited.

Table II. List of some of the implemented Rewriting Rules.

Rule	*Versions*
detectPE3×1BA	PEs of type PE3×1BA
detectPE3×1BB	PEs of type PE3×1BB
detectPE3×1Shift	PEs of type PE3×1Shift
detectPE[4×1,4×2,4×4]	PEs of type PE4×1, PE4×2 or PE4×4
detectPE4×2 usingOnly1Output	PEs of type PE4×2 with one unused output ALU
detectPE4×4 usingOnly1Output	PEs of type PE4×4 with one unused output ALU
detectPE4×4 subsetBothNOOP	PEs of type PE4×4 with both output ALUs doing no operation. Maps two ALU of type B operations into a PE4×4
detectPE4×4 subsetThreeNOOP	A single ALU of a PE4×4
detectMUL-ADD	PEs of type multiply and add
detectADD-MUL	PEs of type ADD-MUL
detectCounter	Maps SSA instructions responsible to control loop iterations

As aforementioned, in order to preserve functionality, the TRS step includes instructions that can be redundant. Suppose the two examples shown in Figure 6. In the case A, the instruction 2 must be preserved since variable “3.0” is used by instruction 5 and the PE3x1 used has only one output and no way to output both variables “3.0” and “5.0” (in this

case, variable “3.0” has only the scope of the PE). Case B uses PE3x2, which can output both variables and thus instruction 2 is removed from the final SSA form.

Table III. Samples of Rewriting Rules.

```
…
ALU Types Definition:
Opcode2a include IADD, ISUB
Opcode2b include IADD, ISUB, IMUL
…
Variable Definition:
opa1, opa2 are of type Opcode2a
opb1, opb2 are of type Opcode2b
c1, c7 are of type Constant
v1, v2, v3, v4, v5 are of type Variable
e1 are of type Expression
…
Rules detectPE3x1BA:
[opb1 | v1, v2 | v3]; e1; [opa1 | v3, v4 | v5]  =>
                         e1; [PE3x1BA | v1, v2, v4| v5
                 {[opb1 | v1, v2 | v3], [opa1 | v3, v4| v5]}]

[opb1| v1, v2| v3]; e1; [opa1| v4, v3| v5]=>
   e1; [PE3x1BA | v1, v2, v4 | v5{[opb1| v1, v2| v3], [opa1| v4, v3|
                                                            v5]}]
…
```

Table IV. List of some of the Implemented Rewriting Strategies.

Strategy	*Rules applied*
PE3×1BA	detectPE3×1BA
PE3×1BB	detectPE3×1BB
PE3×1Shift	detectPE3×1Shift
PE4×1	detectPE4×1
PE4×2	detectPE4×2
PE4×4	detectPE4×4
PE4×2 usingOnly1Output	detectPE4×2 using Only1Output
PE4×2all	detectPE4×2; detectPE4×2 Only1Output
PE4×4all	detectPE4×4; detectPE4×4 Only1Output; detectPE4×4 subsetBothNOOP; detectPE4×4 subsetThreeNOOP
MUL-ADD	detectMUL-ADD
ADD-MUL	detectADD-MUL
Counter	detectCounter
All	All the above plus balanceMul; balanceAddwithCloning

The step performed to remove SSA instructions is very simple since we are in the presence of a representation with static single assignments. This step only needs to determine if uses of variables are reached by definitions/assignments inside groups of instructions (note that some assignments in groups of instructions have inner scope). In such cases, the redundant instructions are removed. After grouping instructions, the variables with only the group scope (internal) are renamed by concatenating the label “.fu#”, where # represents the group number they refer to.

The SSA form with instructions merged in new ones is not adequate when instructions are grouped across basic block boundaries. In this case, we need to preserve the original location of instructions with additional information identifying the template they belong. Below is a segment of the final format for the example in Figure 3.

```
11: [ sLd | 7.1, 1.0 | 15.1 ];
12: [ IMUL | 15.1, 14.0 | 14.1.fu1 ]      [PE3×1_1 #1];
13: [ IADD | 14.1.fu1, 6.1 | 6.2 ]        [PE3×1_1 #2];
14: [ IADD | 1, 7.1 | 7.2 ];
```

This final representation preserves the initial behavior, adds information about the template matching as annotations, and supports templates with instructions across basic block boundaries (e.g., counters for loop control).

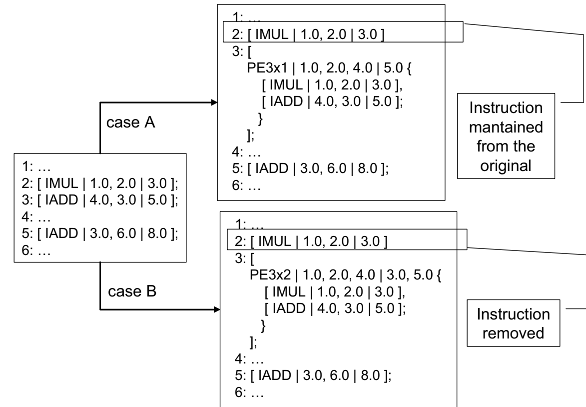


Figure 6. Two examples where an instruction maintained from the original SSA form should be removed (case B) or not (case A).

6. Experimental Results

Table V shows the benchmarks used in our experiments. They resemble typical signal and image processing computational intensive tasks using integer and fixed-point arithmetic. Their SSA complexity ranges from 13 to 210 instructions (an average of about 67 instructions). Figure 7 shows the average percentage of each SSA instruction in those benchmarks. Instructions of type PHI# represent phi SSA form instructions [3] where the integer represents the number of inputs. We can see that integer addition (IADD) is the most used operation (about 22%). The other most represented are IMUL (integer multiplication) and ASSIGN (e.g., assignment of a constant to a variable).

Next we show results on applying our approach to the examples considering the templates illustrated in Figure 5. Table VI shows the average percentage of each of the templates in the benchmarks used. Notice that when coarse-grained architectures have a direct support for that templates in the PEs, they are usually able to execute the other type of instructions presented in the SSA form or other sub-

templates used in this paper (e.g., PEs with direct support for templates of type PE4×1 are usually also able to perform MUL-ADD).

Table V. Benchmark characteristics (TI: Texas Instruments; MB: MediaBench).

Benchmark	Repository	Description	#SSA inst.
fdct	TI	Fast DCT (Discrete Cosine Transform)	210
fft	TI	Fast Fourier Transform	70
fir	TI	Finite Impulse Filter (1D)	17
cplx	TI	Finite Impulse Filter using complex arithmetic (1D)	43
autoc	TI	Auto-correlation (1D)	17
fwt2D	-	Forward Haar Wavelet	84
hamming	-	Hamming encoder	35
smooth	-	Smooth image filter (using a 3x3 window)	34
edge	-	Edge detection	87
sad	-	Sum of absolute differences (1D)	13
adpcm_dec	MB	ADPCM decoder	55
adpcm_enc	MB	ADPCM encoder	78
dct	-	8x8 DCT based on matrix operations (non-optimized)	136
skinDetect	-	Identification of the pixels related to skin in an image	41

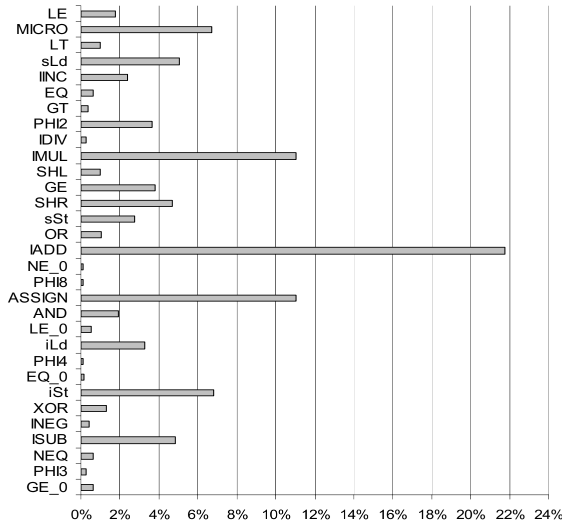


Figure 7. Average percentage of original SSA instructions in the considered benchmarks.

Figure 8 shows the usage percentage of each template for each benchmark being used. As can be seen from the results, the template PE3×1BB is the one with the highest coverage in the benchmarks used. A 13.7% average usage has been identified for this type of template. Concerning PE4×1 and PE4×2 average coverage they seem to be low (2.3% and 0.2%, respectively) but they are over 8% and 10% for the *fdct* example.

As an example of grouping of SSA instructions across basic blocks, the matching of counter type templates has resulted in an average coverage of 4.5% for the benchmarks being used (a maximum result of 11.8% has been achieved for the smooth example). These results were expected since the counters are directly related to loops in the code.

Table VI. Average usage percentage of some templates in the considered benchmarks.

Template	MUL-ADD	ADD-MUL	PE 3:1 BB	PE 3:1 BA	PE3:1 BA Shift	PE 4:1	PE 4:2	PE 4:2 +4:1	PE 4:4
Average (%)	8.0	2.1	13.7	12.1	0.6	2.3	0.2	2.3	0.2
Standard Deviation	7.1	3.4	9.8	10.4	1.7	1.3	0.1	3.1	0.8

7. Related Work

Diverse tools and approaches have been used to program coarse-grained architectures. The PACT XPP [8] coarse-grained array is programmed using the low-level Native Mapping Language (NML). A higher level of abstraction is offered by the XPP-VC Compiler [14]. However, the compiler relies on a subsequent mapping step to bind operations to the PEs of the architecture.

The notion of retargetable compiler has been used in a limited extent when targeting coarse-grained arrays. Such tools have been usually specific to certain architectures with some variations that permit to target a set of architectures preserving common features. Examples are the DRESC Compiler [15] and the KressArray Xplorer [16], which provides a dataflow compiler and a complete system for hardware design space exploration (based on KressArray features).

There has been a renewed interest on VLIW type architectures [1]. Architectures with different PE complexities are being studied and algorithms to identify the sets of operations grouped to each PE being proposed [17][18]. Note, however, that there are three types of approaches. One approach synthesizes suitable architectures (usually based on a template) for a set of benchmarks and thus needs to identify the best template instructions. A second one considers the existence of certain architectures and maps sets of operations into the PE's structures (which implement distinct templates). A third approach tries to expose custom instructions from sets of instructions or from dataflow graphs that represent a source program. For that, template matching is usually used [19]. In our work we are focusing more on the second approach being, however, able to retarget different architectures and to exploit some PE's characteristics.

Term Rewriting Systems and Rewriting Logic have been recently used in a number of applications, especially in the context of prototyping algebraic operations in reconfigur-

able systems [20], verification of arithmetic circuits [21] and hardware synthesis [22]. Rewriting logic has been shown to have greater flexibility than pure rewriting for the discrimination between fixed and reconfigurable elements of reconfigurable architectures, allowing for a natural and quick conception and simulation of implementations of new reconfigurable computing paradigms. In this context, Ayala et al. used rewriting logic for modeling the reconfiguration of dynamically reconfigurable architectures [23]. Morra et al. applied rewriting-logic to the generation of functionally equivalent implementations of mathematical functions in reconfigurable hardware; their tool flow for design space exploration and application examples are presented in [24][25].

The use of Rewriting Logic and Term Rewriting Systems explained in this paper is to the best of our knowledge new.

8. Conclusions

This paper presents the use of Rewriting Logic for template matching and algebraic optimizations, bearing in mind the mapping of imperative programming languages to coarse-grained reconfigurable architectures. The approach uses a three-address code SSA (static single assignment) representation of the kernel being mapped and identifies sets of SSA instructions, each set suited to be executed by a single processing element of the target architecture. The concept can be used for retargetable compilation and/or for design space exploration in the context of coarse-grained reconfigurable architectures.

Ongoing work intends to extend the approach with exploration of the strategies that must be applied to a better decision on the template coverage. A high-level model, able to acquire the main characteristics of the target architectures being exploited, is under development in order to estimate the latency to execute each kernel and to serve as a figure of merit for design decisions.

Acknowledgements

This work has been possible due to the bilateral DAAD/CRUP cooperation project entitled “Architecture and Compilation Exploration for a Dynamically Reconfigurable System-on-Chip (ACER)”. João Cardoso would like also to acknowledge the support of the project CHIADO, funded by the Portuguese Foundation for Science and Technology (FCT), POSI and FEDER.

References

[1] K. Fan, M. Kudlur, H. Park, and S. Mahlke, “Compiler-directed Synthesis of Multifunction Loop Accelerators,” in *Workshop on Application Specific Processors (WASP)*, Sep. 2005, pp. 91-98.

[2] R. Hartenstein, “A Decade of Reconfigurable Computing: a Visionary Retrospective,” In *Int’l Conf. on Design, Automation and Test in Europe (DATE)*, Munich, Germany, March 12-15, 2001, pp. 642-649.

[3] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA, 1997.

[4] R. Rodrigues, and João M. P. Cardoso, “On Pipelining Sequences of Data-Dependent Loops,” in *Journal of Universal Computer Science (JUCS)*, to appear.

[5] J. M. P. Cardoso, “CHIADO: compilation of high-level computationally intensive algorithms to dynamically reconfigurable computing systems,” in *SPIE Microtechnologies for the New Millennium 2005 Symposium*, Seville, Spain, May 9-11, 2005, SPIE Vol. 5837, pp. 893-901.

[6] H. Singh et al., “MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications,” *IEEE Trans. on Computers*, Vol. 49, no. 5, May 2000, pp. 465-481.

[7] B. Mei, A. Lambrechts, D. Verkest, J.Y.s Mignolet, R. Lauwereins, “Architecture Exploration for a Reconfigurable Architecture Template,” in *IEEE Design & Test of Computers*, 22(2), 2005, pp. 90-101.

[8] V. Baumgarte, et al., “PACT-XPP – A Self-reconfigurable Data Processing Architecture,” In *Journal of Supercomputing*, Kluwer Academic Publishers, vol. 26, issue. 2, September 2003, pp. 167-184.

[9] M. D. Galanis, G. Theodoridis, S. Tragoudas, C. E. Goutis, “A High Performance Data-Path for Synthesizing DSP Kernels,” in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 25, no. 6, June 2006, pp. 1154-1163.

[10] F. Baader, and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[11] P. Borovansky, C. Kirchner, H. Kirchner, and P. Moreau, “ELAN from a rewriting logic point of view,” *Theoretical Computer Science*, vol. 285, no. 2, 2002.

[12] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C. Talcott. “The Maude 2.0 System” in *Rewriting Techniques and Applications (RTA 2003)*, LNCS 2706, Springer-Verlag, June 2003, pp. 76-87.

[13] J. Meseguer, N. Martí-Oliet and A.Verdejo. “Towards a strategy language for Maude” in *Proc. Fifth Int’l Workshop on Rewriting Logic and its Applications (WRLA 2004)*, Electronic Notes in Theoretical Computer Science, Elsevier, 2004.

[14] J. M. P. Cardoso, and M. Weinhardt, “XPP-VC: A C compiler with temporal partitioning for the PACT-XPP architecture,” in *Proc. 12th Int’l Conf. on Field-Programmable Logic and Applications (FPL’02)*, 2002.

[15] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, “DRESC: A retargetable compiler for coarse-grained reconfigurable architectures,” in *Proc. Int’l Conference on Field Programmable Technology (FPL’02)*, 2002.

[16] R. W. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, “KressArray Xplorer: a new CAD environment to optimize reconfigurable datapath array,” in *Proc. ASP-DAC*, vol. 1, 2000, pp. 163-168.

[17] H. Park, K. Fan, M. Kudlur, and S. Mahlke, “Modulo Graph Embedding: Mapping Applications onto Coarse-Grained Reconfigurable Architectures,” in *Proc. Int’l Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES’06)*, Oct. 2006.

[18] N. Clark, H. Zhong, and S. Mahlke, “Automated Custom Instruction Generation for Domain-Specific Processor Acceleration,” in *IEEE Transactions on Computers*, Vol. 54, No. 10, Oct. 2005, pp. 1258-1270.

[19] R. Kastner, A. Kaplan, S. Ogrenci Memik, and E. Bozorgzadeh “Instruction generation for hybrid reconfigurable systems,” in ACM *Trans. Design Autom. Electr. Syst. (TODAES)*, 7(4), 2002, pp. 605-627.

[20] M. Ayala-Rincón, Carlos Llanos, Ricardo P. Jacobi, Reiner W. Hartenstein, “Prototyping Time and Space Effiicient Computations of Algebraic Operations over Dynamically Reconfigurable Systems Modeled by Rewriting-Logic,” in *ACM Transactions On Design*

Automation Of Electronic Systems (TODAES), vol. 11, no. 2, 2006, pp. 251-281.

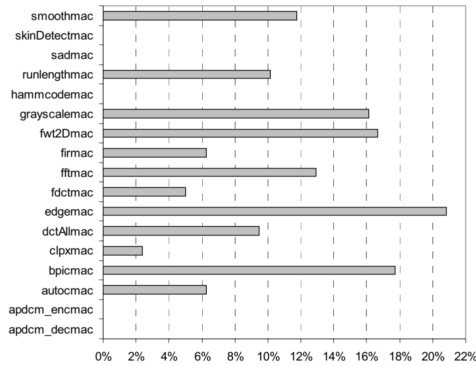
[21] D. Kapur and M. Subramaniam, “Using and induction prover for verifying arithmetic circuits,” *Journal of Software Tools for Technology Transfer*, vol. 3, no. 1, pp. 32–65, Sept. 2000.

[22] Arvind and X. Shen, “Using term rewriting systems to design and verify processors,” *IEEE Micro*, vol. 19, no. 3, 1999, pp. 36–46.

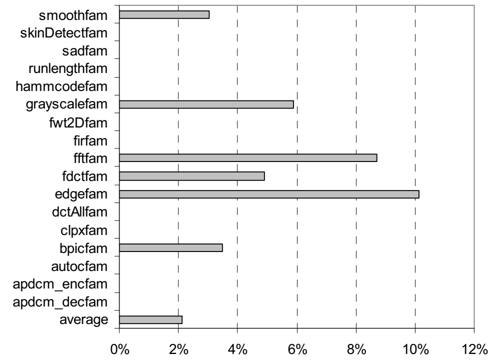
[23] M. Ayala-Rincon, R. Jacobi, L. Carvalho, C. Llanos, and R. Hartenstein, “Modeling and prototyping dynamically reconfigurable systems for efficient computation of dynamic programming methods by rewriting-logic,” in *Proc. SBCCI’04*, 2004.

[24] C. Morra, J. Becker, M. Ayala-Rincon, R. Hartenstein. “FELIX: Using Rewriting-Logic for Generating Functionally Equivalent Implementations” in *Proc. 15th Int’l Conference on Field-Programmable Logic and Applications (FPL’05)*, Aug. 24 - 26, 2005, Tampere, Finland.

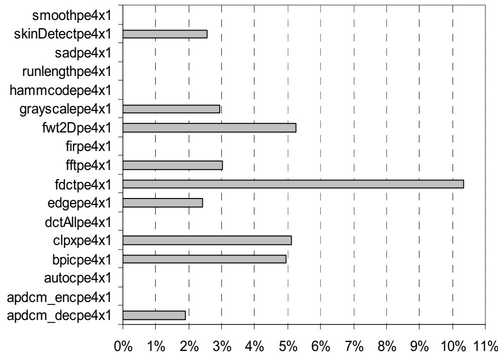
[25] C. Morra, M. Sackmann, S. Shukla, J. Becker, R. Hartenstein. “From Equation to VHDL: Using Rewriting Logic For Automated Function Generation,” in *16th Int’l Conference on Field-Programmable Logic and Applications (FPL’06)*, Aug. 28 - 30, 2006, Madrid, Spain.



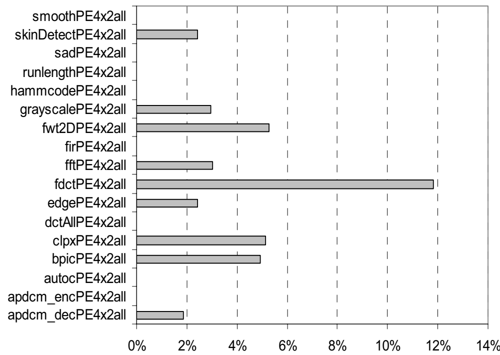
(a) MUL-ADD



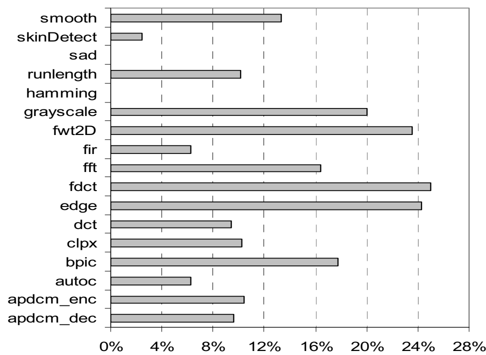
(b) ADD-MUL



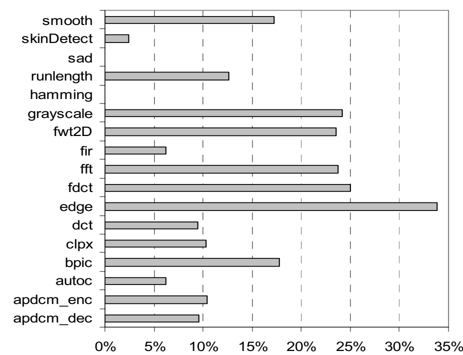
(c) PE4x1



(d) PE4x2:4x1



(e) PE3x1AB



(f) PE3x1BB

Figure 8. Usage percentage for a number of templates in the presented benchmarks.