Extensible Pattern Recognition in DSP Programs using Cetus

Amin Shafiee Sarvestani
Dept. of Computer and Information Science,
Linköping University, Sweden
amish805@student.liu.se

Erik Hansson
Dept. of Computer and Information Science,
Linköping University, Sweden
erik.hansson@liu.se

Christoph Kessler
Dept. of Computer and Information Science,
Linköping University, Sweden
christoph.kessler@liu.se

***Abstract*—We describe a tool for pattern (idiom) recognition in DSP (digital signal processing) source programs. We focus on patterns for loops and the statements in their bodies as these often are the performance-critical constructs in DSP applications for which replacement by highly optimized, target-specific parallel algorithms will be most profitable. For better structuring and efficiency of pattern recognition, we classify patterns by different levels of complexity such that patterns in higher levels are defined in terms of lower level patterns.**

Our tool utilizes functionality provided by the Cetus compiler infrastructure for static pattern recognition. It works on the Cetus intermediate representation (IR) and applies bottom-up pattern matching. For better extensibility and abstraction, most of the structural part of recognition rules is specified in XML form. This separates the tool implementation from the pattern specifications.

I. Introduction

Modern special-purpose chip multiprocessor architectures designed for special-purpose application areas such as digital signal processing (DSP) are highly optimized, heterogeneous and increasingly parallel systems that must fulfill very high demands on power efficiency. Architectural features include advanced instructions, SIMD computing, explicitly managed on-chip memory units, on-chip networks and reconfiguration options, all of which are exposed to the programmer and compiler. One example is the ePUMA architecture [2] being developed at Linköping University, a low-power high-throughput multicore DSP architecture designed for for emerging applications in mobile telecommunication and multimedia.

This high architectural complexity makes it very hard for programmers and especially for compilers to generate efficient target code if starting from (even well-written) sequential legacy C code. Domain-specific languages such as SPIRAL [8] are one possible way of enabling efficient platform-specific code generation but require rewriting of the program code from scratch in a new language. In this work, we propose instead an approach for automatic porting of applications by statically analyzing their source code for known frequently occurring programming idioms captured as *patterns*, and replacing recognized code parts by an equivalent implementation that is highly optimized for the target architecture, such as library code or code generated from autotuning tools such as SPIRAL.

For our work as presented here, we started from an earlier approach for pattern recognition in scientific source codes, the PARAMAT approach [4]. A *pattern* is an abstraction of a computation that generally can be expressed in many different ways using a given programming language such as C, even if some restrictions (such as absence of pointers) apply. By specifying known recognition rules for patterns in terms of language elements such as for loops, assignment statements and expression operators, a tool can, as far as enabled by the specified rules, identify occurrences of patterns in source code. Recognition is conservative, i.e. a pattern matches only if one of its recognition rules completely applies; if some constraint in a recognition rule can not be checked statically, the entire rule fails. If the patterns and their recognition rules are carefully defined, the matching process is deterministic. This implies that any given snippet of source program code can match at most one pattern.

Our tool initially uses the same principle and matching algorithm as PARAMAT [4] and also re-uses most of the patterns specified in that earlier work. Our tool is however much more modular and extensible. It uses an object oriented language, allowing for a modern OO design for the specification of patterns and use of e.g. reflection for convenient expression of recognition rules. We also developed an XML format for specifying patterns and the structural parts of their matching rules; this pattern specification parameterizes the matching tool, which separates the specification of recognition rules from the implementation of the matching tool itself, as no pattern is hardcoded and the list of patterns can be extended independently from the tool. Some of the patterns adopted from PARAMAT are modified and some new patterns are defined to also cover (part of) the DSP domain.

Moreover, our tool is built on top of Cetus [3], a modern industrial-strength compiler framework with better support for source-level program analysis and transformation.

The remainder of this paper is organized as follows. Section II gives a very simple example of patterns, the recognition process and the hierarchial structuring of the set of patterns. In Section III we give an overview of the architecture and how the pattern matching mechanism works. Section IV gives more details about how Cetus is used in the implementation, and Section V presents first experimental results for the tool. We conclude with a short review of related work and give an outlook to future work.

II. Patterns and pattern hierarchy

Many of the patterns that are candidates for replacement by optimized target-specific code are characterized by (for) loops. As recognition works step-wise bottom-up and is conservative, all statements in a loop body must be matched by some pattern before the entire loop can be recognized. Hence, we also need to define patterns for elementary program constructs such as constants or variable and array accesses, operators and intrinsic functions, expressions and assignments.

For example, a simple assignment of a constant to a variable such as

```
x=c;
```

is easily recognized as an occurrence of a pattern called *SINIT* (scalar initialization). The assignment statement is annotated with a *pattern instance* which could (e.g., for debugging purposes) be shown as annotation of the program code in the following way:

```
// SINIT(x,c)
x=c;
```

Recognition rules are formulated to match the bottom-up recognition process. Hence, rules for higher level patterns are defined in terms of instances of lower level patterns. For example, if the *SINIT* pattern occurs inside a for-loop, we could find an instance of the *VINIT* pattern (vector initialization) as in the following example

```
// VINIT(_i0,_x0,c)
for (i=K; i<n; i+=step)
  // SINIT(x[i],c)
  x[i]=c;
```

where `_i0` and `_x0` are newly created symbols constructed by the recognizer that denote a loop range object and a vector container, respectively, which internally hold the parameters about loop and access bounds etc.

For each pattern p, there exists a list of next-level patterns or "superpatterns" which can be built from an occurrence of p. This relationship among the patterns is shown in a graph called the *pattern hierarchy graph* (PHG). Figure 1 shows a small excerpt of this graph with some simple patterns.

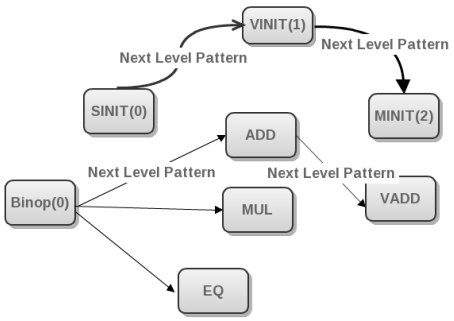

Fig. 1. Excerpt of the Pattern Hierarchy Graph

III. Recognition method and tool design

A. Pattern Detection

The recognition tool works on top of the Cetus IR tree, which is generated from the C source file of a DSP program. This tree is traversed in post-order, using the algorithm presented in [4].

Leaf nodes are of type *Identifier* or *Literal* in Cetus. Identifiers and Literals are trivial patterns but are defined and recognized as they are the basis for the definition and recognition of other patterns.

For recognizing the root node of any subtree, the tool inspects the PHG to get the list of candidate patterns that can be built atop the previously detected patterns for the children nodes. The structural constraints in the recognition rules of each candidate pattern are then checked against the list of detected patterns in the children of this subtree. If any pattern matches, a new data structure is created which contains a reference to the current subtree and another one to the instance (summary data structure) of the detected pattern. This structure is kept for the parent node of the newly matched subtree in a hash table. If the matching fails for all candidates, this subtree is no longer considered and the matching process continues in other branches. This process continues until the whole tree is traversed.

For certain patterns and rules, some extra checking beyond the structural constraints must be applied as well; for these there exists a specified function, which is uniquely defined for each pattern. After initial checking of the tree structure, this specified function is loaded by reflection and takes care of the extra checking required for that particular pattern.

Figure 2 shows the matching process at the root node of the illustrated subtree. The root node is a for loop header with only one child, which already has been detected as occurrence of a *SINIT* pattern. When the tool reaches the root node the information regarding the patterns detected in children nodes are retrieved from a hash table, then this information (here, the *SINIT* pattern) is sent to the PHG to find the candidates for the next level patterns defined on top (in this case, e.g., *VINIT* and *VASSIGN* patterns). The candidates are sent to the pattern matcher, which compares the internal structure of each pattern against the structure of the parent node and its children.

B. Pattern definition

All patterns are defined as XML nodes in a pattern specification file. For each pattern there exists an XML element that describes the name and the structural part of recognition rules.[1] As an example, Figure 3 shows the definition of the *SINIT* pattern. The expected structure of each pattern is defined based on the IR node type names used in the Cetus IR. The "structure" tag describes the expected number of elements and their node types. For example, Figure 3 shows that *SINIT*

[1]In the current implementation we also need to specify the list of superpatterns to build the PHG; this information will be extracted automatically from the recognition rules in the XML file in a future version of our tool.

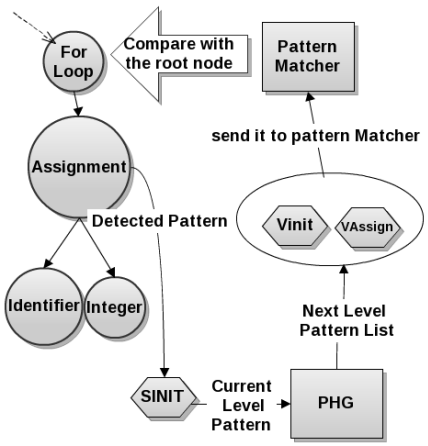

Fig. 2. The matching process at the root based on the children's patterns

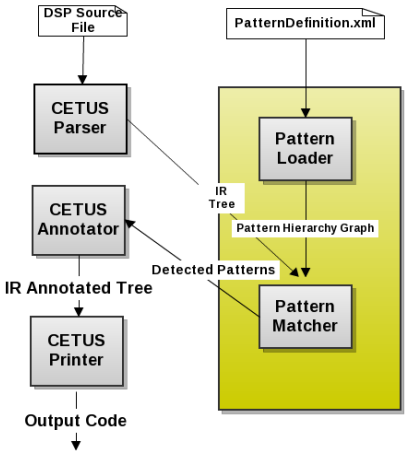

Fig. 4. Pattern Recognizer Architecture

has two elements, where the first one is either *Identifier* or *ArrayAccess* and the second one can be either *FloatLiteral* or *IntegerLiteral*.

Before the matching process starts, the XML file is parsed and the PHG data structure (with patterns, recognition rules etc.) is generated.

```
<basepattern    name="SINIT"  >
<roottype>AssignmentExpression</roottype>
<structure>
 <element orderid="1">
  <type>Identifier</type>
  <type>ArrayAccess</type>
 </element>
 <element orderid="2">
  <type>FloatLiteral</type>
  <type>IntegerLiteral</type>
 </element>
</structure>
...
</basepattern>
```

Fig. 3. Definition of the *SINIT* pattern in the XML-based specification.

C. System architecture

The general architecture of the system can be seen in Figure 4. The input program source file is parsed by Cetus and the IR tree is generated. Our tool builds the PHG data structures from the XML specification file, which are then passed to the pattern matcher to start the matching process. The final results are sent to the Cetus annotator which annotates each node by its equivalent pattern.

D. Extensibility

The extension by new patterns can be accomplished easily as they are defined independently of the recognition tool. Each new pattern is added by defining the structure inside a new XML tag. If the pattern requires extra checking, we only need to implement that specified function, which would be later automatically called by reflection.

E. Horizontal pattern matching

Beyond vertical pattern matching as described above, there are also patterns that cover (a merge of) several sibling nodes and thus require horizontal pattern matching [4]. For example, three *SINIT* pattern instances in the following code can be merged into an instance of a *SWAP* pattern on variables `x` and `y` using `z`:

```
z = x;          // SINIT(z,x)
x = y;          // SINIT(x,y)
y = z;          // SINIT(y,z)
```

Before the pattern recognition process at the parent level of any subtree starts, the list of the children patterns are independently examined for horizontal patterns in order to see if they can be merged. If yes, the IR nodes with the old pattern instances are then deleted (inactivated) and replaced (new node inserted to carry the newly detected horizontal pattern). Details of guiding horizontal matching by data flow edges can be found in [4].

IV. Details of interaction with Cetus

The recognition tool was built on top of Cetus v1.2.1, which was the latest version at the time of starting the development. The package `cetus.hir` provides most of the functionality needed for our tool. The key to the smooth implementation of the tool was the Cetus IR tree. It provided an efficient and high level tree structure which could be easily traversed. However, modifying the tree was not as easy as expected, and it generated some issues in certain cases. The `cetus.hir` package offered lots of functions that were specific based on the type of the node which facilitated the process of accessing the internal structure of each object. This was especially important as we have lots of patterns whose definitions depend on the for loop construct, but using the package we could easily access its parts such as the initial statement, the loop step and the statements of the loop body.

File Name	Size	Time	Patterns	Rec. Rate
AverageFilter.c	2.2 KB	0.239 s	129	35%
GaussianFilter.c	1.5 KB	0.204 s	45	79%
RedColorEnhancement.c	2.6 KB	0.333 s	105	54%
FourPointDCT.c	0.6 KB	0.17 s	33	84%

TABLE I
EVALUATION OF THE TOOL WITH FOUR DSP SOURCE CODES.

After the pattern detection phase has completed, we display the result by annotating each subtree root in the IR by its equivalent detected pattern (if any). For this purpose, each pattern implements a *print* function that generates an annotation in string format. Our tool uses the annotation features in Cetus and adds the generated annotation string to the specified subtree. The only thing left is to call the *print* function provided by Cetus, which will print both the code and the annotations, such that the pattern instance is seen as a comment statement preceding each matched statement, as shown earlier.

The expression simplifier functionality of Cetus is another useful feature as we need to simplify some certain statements before applying the pattern recognition tool.

The iterators implemented by Cetus offers a *Next* function, which gets the next node in the tree traversal based on the order of the iterator. However, the relationship between the current node and the next node provided by the *Next* function is unknown. Our tool applies different rules depending on the relationship between the current and the next node, so knowing whether the next node is the sibling or part of another branch makes a big difference, while the iterator structure does not provide this kind of information unless checked manually.

V. EVALUATION

For an early evaluation of the pattern detection rate and performance, we applied the first prototype of our tool to several DSP source files taken from [5]. We were especially interested in the ratio of the time spent inside the recognizer tool to the file size and the number of detected patterns. The execution time of the tool is expected to increase with the size of the source code; however, the structure of the code itself can affect the execution time. A certain source code might contain more instances of defined patterns and consequently execution time increases, Table I shows the results for some example DSP source codes. The time shown includes the time taken by the pattern recognizer and the time taken by Cetus to print the output code.

The recognition rate is given as the percentage of matched statements.[2] The differences in the recognition rate in Table I originate from the code structure. GaussianFilter and FourpointDCT contain a large percentage of low-level patterns that are fully implemented; as a result, the high recognition rate was expected for these files.

[2]The percentage of matched for loops may be another good metric for the recognition rate, which we will use in future work.

VI. RELATED WORK

Pattern recognition and automatic parallelization has been researched extensively. For instance, Kessler [4] and Di Martino [6] describe recognition tools for automatic parallelization. Pottenger and Eigenmann [7] implemented a Fortran idiom recognizer in Polaris, which is a predecessor of Cetus. The XARK compiler [1] focuses on recognizing a collection of computational kernels and works on a special high level IR known as Gated Single Assignment (GSA) form. We refer to [1] for further references.

VII. FUTURE WORK

This is work in progress. We are continuously adding more higher level patterns and plan to analyze a larger set of example codes.

At this point we restrict our input programs to be pointer free due to the complexity of analyzing them statically. Future work may also include a technique to integrate and detect patterns involving pointers.

Our pattern recognition tool is planned to be part of a compiler tool chain for the low-power high-throughput multicore DSP architecture ePUMA that is currently being developed at Linköping University for emerging applications in mobile telecommunication and multimedia [2]. The pattern information derived by the tool will help with optimized ePUMA code generation by automatically replacing the (nontrivial) detected pattern instances with expert-written computation kernels that are highly tuned for ePUMA. For this purpose, an alternative source code emitter will be written that emits function calls instead of annotations in the original source code.

Acknowledgements This work was partly supported by SSF.

REFERENCES

[1] Manuel Arenaz, Juan Touriño, and Ramon Doallo. Xark: An extensible framework for automatic recognition of computational kernels. *ACM Trans. Program. Lang. Syst.*, 30:32:1–32:56, October 2008.

[2] Erik Hansson, Joar Sohl, Christoph Kessler, and Dake Liu. Case study of efficient parallel memory access programming for the embedded heterogeneous multicore DSP architecture ePUMA. In *Proc. Int. Workshop on Multi-Core Computing Systems (MuCoCoS-2011), June 2011, Seoul, Korea. IEEE CS Press*, 2011.

[3] Troy A. Johnson, Sang ik Lee, Long Fei, Ayon Basumallik, Rudolf Eigenmann, and Samuel P. Midkiff. Experiences in using Cetus for source-to-source transformations. In *Proc. 17th Int. Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2004.

[4] Christoph W. Kessler. Pattern-driven automatic parallelization. *Scientific Programming*, 5(3):251–274, 1996.

[5] H. Malepati. *Digital Media Processing: DSP Algorithms Using C*. Elsevier Science, 2010.

[6] Beniamino Di Martino and Giulio Iannello. Pap recognizer: A tool for automatic recognition of parallelizable patterns. In *Proc. 4th Int. Workshop on Program Comprehension*, Los Alamitos, CA, USA, March 1996. IEEE Computer Society.

[7] Bill Pottenger and Rudolf Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. In *Proc. 9th Int. Conf. on Supercomputing*, pages 444–448. ACM, July 1995.

[8] Markus Püschel, Jose M. F. Moura, Jeremy R. Johnson, David Padua, Manuela M. Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicolas Rizzolo. Spiral: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2), February 2005.