OORS: An Object-Oriented Rewrite System with Applications in Retargetable Code Generation and Optimization

Gernot Gebhard and Philipp Lucas

Saarland University, Dept. 6.2 Computer Science,
Building E1.3, Compiler Design Lab,
D-66041 Saarbrücken, Germany
[gebhard|phlucas]@cs.uni-sb.de,
Website: http://rw4.cs.uni-sb.de/~gebhard/projects/oors/

Abstract. Retargeting a compiler's back end to a new architecture is a time-consuming process. This becomes an evident problem in the area of programmable graphics hardware (graphics processing units, *GPUs*) or embedded processors, where architectural changes are faster than elsewhere. We propose the *object-oriented rewrite system OORS* to overcome this problem. Using the OORS language, a compiler developer can express the code generation and optimization phase in terms of cost-annotated rewrite rules supporting complex non-linear matching and replacing patterns. *Retargetability* is achieved by organizing rules into profiles, one for each supported target architecture. Featuring a rule and profile inheritance mechanism, OORS makes the reuse of existing specification possible. This is an improvement regarding traditional approaches. Altogether OORS increases the maintainability of the compiler's back end and thus both decreases the complexity and reduces the effort of the retargeting process. To show the potential of this approach, we have implemented a code generation and a code optimization pattern matcher supporting different target architectures using the OORS language and introduced them in a GPU compiler.

1 Introduction

As the number of different hardware architectures is steadily growing, easily retargetable compilers are most valuable. For instance, this applies to graphics processing units (GPUs). For the last few years the performance of GPUs has been increasing at a much faster rate than that of general-purpose processors and now exceeds the peak performance of high-end CPUs. The amount of transistors on graphics chips has been growing by a factor of 32 every two years [2]; 16 times faster than the CPU transistor count growth, which Moore's Law predicts. Thus, GPUs have become more and more interesting for general-purpose programming [17]. Several high-level languages, such as Brook for GPUs [3] or CGiS [14, 15], have emerged to exploit the vast computational power that GPUs have to offer. Easily retargetable compilers for these languages are necessary, because new

GPU architectures are released at a fast rate (e. g., NVIDIA's NV40 in 2004, G70 in 2005, and G80 in 2006). Naturally, the same applies to compilers supporting embedded systems, where a wide variety of different architectures with common heritage exists (e. g., Freescale's MPC555, MPC565, MPC755 and derivatives).

To decrease the complexity of the retargeting process and to keep the compiler maintainable in the long run, we propose the *object-oriented rewrite system OORS*. The idea of OORS originated from the fact that new architectures share many features with their predecessors, but offer an extended instruction set (apart from other new features). If a new architecture is released with minor differences to an already supported one, only small changes to an existing back end are required. So, the key feature of our approach is to be found in the *reusability* of existing specifications. We realized this by introducing object-oriented language features in the bison-like OORS language. Easily making the reuse of existing specifications possible, the proposed language enables a compiler developer to implement an OORS code generator or code optimizer which is less complex than a hand-written one. Consequently, the OORS implementation is much easier to read and to maintain in the long run. Our experiences in introducing new GPU architectures (NV40/G80) in the CGiS compiler supports this claim.

Basically, an OORS specification describes a pattern matcher that translates attributed input strings into attributed output strings. Given a set of rewrite rules, the pattern matcher processes the input string as follows. First the pattern matcher tries to match the possibly non-contiguous pattern of each rewrite rule against the input string. If multiple rules are applicable, the pattern matcher computes the non-constant costs of each rule to determine the rule to apply. Finally, the selected rule emits an attributed string that is appended to the output string. After the whole input string has been consumed, the pattern matcher terminates.

The aim of this paper is to introduce the OORS language, its syntax and semantics and to demonstrate its applicability in real word applications. Additionally, this paper briefly introduces the pattern matcher generator OORG that compiles an OORS specification into a C^{++} *dynamically retargetable* pattern matcher. We show its applicability by means of a real-life compiler for GPUs.

The remainder of this paper is structured as follows: Section 2 discusses related work. The object-oriented rewrite system OORS is introduced in Section 3. This section covers both the OORS language and the matching process. Section 4 introduces the pattern matcher generator OORG and demonstrates the integration of an OORS pattern matcher in a compiler. Section 5 concludes this paper and discusses future work.

2 Related Work

Numerous other approaches have been suggested for code generator generators. Below we discuss a representative set of such work. More systems are mentioned in [12].

Emmelmann et al. [7] propose BEG, a generator for efficient back ends. Using the description language BEGL, the developer implements tree-pattern matchers for code generation in terms of cost-annotated rewrite rules. In contrast to the OORS language, BEGL offers no rule-inheritance mechanism. The reuse of existing specifications is thus not possible. Additionally, a BEG pattern matcher is dedicated to a single target architecture only. The major difference is to be found in the processed input data. BEG code generators process *trees*, whereas OORS pattern matchers operate on instruction *sequences*.

In [11] Fraser et al. introduce the code generator BURG for the bottom-up rewrite system BURS which is similar to BEG. BURG is able to generate tree-pattern matchers for fast optimal instruction selection. A BURG-generated tree-parser is able to find an optimal parse of an input tree in linear time. As in BEGL, the BURG grammar does not feature any mechanisms that makes the reuse of existing specification possible. Additionally, BURS code generators are only able to generate code for a single architecture only. BURG-generated code selectors are used in the ANSI C compiler lcc [10].

Ferdinand et al. [9] solve the code selection problem with deterministic finite tree automata that are generated automatically from regular tree grammars. In contrast to BURG, the left-hand and the right-hand side of rules are not limited to leafs or nodes with one or two child nodes. In contrast to OORS, the costs of a rule must be constant and thus cannot depend on the matched instructions. Similar to BURG, the developer cannot inherit rewrite rules from each other to easily copy reusable properties. As in the other approaches discussed above, it is not possible to target multiple architectures. Deterministic tree automata have been successfully tested for several architectures, but are not employed in any existing compiler – at least not to our knowledge.

In [1] Alt et al. propose the CoSy model, which provides a framework for flexible combination and embedding of compiler phases to ease the construction of parallel and optimizing compilers. Using three different languages, the compiler developer can implement the different phases of a compiler on a high level of abstraction. Additionally, the developer is able to specify the control flow and the interactions of the compiler phases. Existing implementations of compiler phases can be simply reused. However, if any modifications to the implementation are required, the framework does not prevent copy-and-pasting of that implementation before the developer can change the code. Just as in the approaches discussed above, the code generator is based on tree-pattern matching. A comparison with OORS is hardly possible, because OORS is designed to only implement the code generation and code optimization phase.

Dias and Ramsey [6] propose a recognizer for machine-independent code selection and code optimization. A recognizer is generated automatically from a declarative machine description that describes properties of the target platform. The generated recognizer requires the compiler to represent intermediate code in the form of machine-independent register-transfer lists (RTLs) [5]. By means of a declarative machine description, the recognizer tries to generate better RTLs. The recognizer will continue until no more optimizations can be applied. The

recognizer omits a previously generated RTL, if the new RTL cannot be implemented on the target platform according to the machine description. The authors have successfully generated and tested a recognizer for the x86 back end in the Quick C-- compiler [4]. This approach differs greatly from OORS pattern matchers, as the developer does not have to explicitly implement the code optimizer. The effort is shifted to implement a complete machine description.

Farfeleder et. al [8] describe a similar approach. By means of a new architecture description language (ADL), the authors are able to derive an optimized tree-pattern matching instruction generator, a register allocator and an instruction scheduler. To demonstrate the applicability of the new ADL, the authors have implemented an ADL-generated compiler for the xDSPcore digital signal processor. Again, the effort is shifted to implement a complete machine description.

In [13] Lerner et al. introduce Rhodium, which is a new language for compiler optimizations, whose soundness can be automatically proven. The developer specifies optimizations in terms of transformation rules that are automatically proven to be semantics-preserving. Rhodium optimizations are not bound to a specific target architecture, because they process input programs transformed into a C-like intermediate language. In this way, the optimizations are automatically retargetable. However, the main goal of their approach are the automated soundness proofs of the compiler optimizations. It is unclear how Rhodium optimizations compete with OORS optimizations, because OORS operates on the instruction level. Apart from that, the approach is out of scope for this paper, because OORS was not designed for automated soundness proofs.

OORS differs in many ways from the approaches presented above. One difference concerns the way in which the *subjects of pattern matching* are represented: OORS operates on sequences of instructions, not on trees. This is because OORS is also employed in the code optimization phase. By representing the subject of matching as an instruction sequence, scheduling properties can be expressed alongside with other low-level optimizations (see Section 3.3). Another important point is that OORS features object-oriented language constructs that make the *reuse* of existing specifications easily possible. Apart from CoSy [1], none of the presented approaches was designed with reusability in mind.

3 Object-Oriented Rewrite System

In this section, we introduce the key concepts of the OORS language and discuss the pattern matching process. For the sake of simplicity, we only concentrate on code *generation*. Section 3.3 briefly discusses the changes to the matching process required to realize code *optimization*.

3.1 Rules

As hinted in Section 1, an OORS pattern matcher processes attributed strings, which are sequences of instructions. We assume that each instruction is an in-

stance of a class of the compiler's internal representation (e. g., a binary instruction could be an instance of the *BinaryInstruction* class) with a common base class. The available attributes of each instruction object are then defined by the member functions of the corresponding class. So to speak, OORS rewrite rules determine transformations on sequences of instruction objects. The behavior of each rule is determined by the following four aspects:

search pattern: The search pattern determines constraints on the structure of the input that must be fulfilled before the pattern matcher may apply the rule. A search pattern is a non-empty, ordered or unordered, possibly discontinuous sequence over instruction classes. By using *wildcard patterns*, the developer can specify search pattern with an arbitrary lookahead.

Each symbol of the search pattern may be guarded by a local side condition, which is simply a boolean expression. Using local side conditions, the developer is able to specify non-linear search patterns. For instance, a local side condition could check whether an operand of the currently matched instruction and the target of a previously matched instruction are of the same type (see Example 3).

condition: The condition corresponds to the local side conditions introduced above. The main difference is that the developer implements a boolean function instead of a single boolean expression. An undefined condition function is assumed to return `true`. Condition functions come in handy, when deriving rules from each other (see Section 3.2).

costs: The cost function associates a weight to each matched instance of the search pattern. The computed costs need not be constant and may depend on the matched instruction objects. If multiple rules match the input sequence, the pattern matcher determines the rule to apply according to the associated costs.

replace pattern: The replace pattern determines the generated instruction sequence that the pattern matcher appends to the output sequence when applying the rule. Each element of the replace pattern corresponds to a constructor call of the instruction class with appropriate arguments. It is possible to access previously generated instructions when initializing a new instruction object.

Simple Rules Example 1 demonstrates how to specify a simple code generation rule that compiles a binary instruction for which the target architecture has a direct counterpart. The rule matches any binary instruction object. Thus, there is no need to implement a rule for every single binary instruction. This keeps the specification both readable and maintainable in the long run.

Example 1 (Rule that compiles any abstract binary instruction into its counterpart). Matched instructions objects can be accessed via the `$$` and `$i`-operators like in yacc and related tools. The `$i`-operators enumerate both the matched and the generated operations uniformly.

```
rule binary {
  search:  [ BinOp ]
  cost:    { return 1; }
  replace: [ GPUBinOp($1->opcode,$1->tgt,$1->op1,$1->op2) ]
}
```

Generic rules like the one shown in Example 1 are not always sufficient: Some instructions may require special rules. Example 2 shows how to compile the exponentiation operator for recent GPU architectures.

Example 2 (Code generation rule for a special unary operator). Instruction sets of recent GPU architectures do not feature an exponentiation operator. Instead, their instruction set contains the `EX2` instruction, which computes powers of two. By using the identity $2^x = e^{\ln(2)x}$, we are able to express e^x on the GPU with the rule `exp`. Note that OORS allows the developer to access generated instructions. This is required to initialize the second generated instruction. In this context, the developer can access the target of the generated multiplication instruction via the `$2`-operator.

```
rule exp {
  search:  [ UnOp($$->opcode == OP_EXP) ]
  cost:    { return 2; }
  replace: [ GPUBinOp(OP_MUL, SymReg(TYPE_FLOAT), $1->op,
                      Constant(TYPE_FLOAT, 1/ln(2))),
             GPUUnOp(OP_EX2, $1->tgt, $2->tgt) ]
}
```

Complex Rules In general simple rules as demonstrated above are not sufficient. In some cases, the developer might want to match instructions objects that are not necessarily adjacent to each other in the input instruction sequence. Example 3 demonstrates a typical case.

Example 3 (Complex code generation rule). GPU architectures feature a combined sine-cosine instruction `SCS` [16]. From a single operand c, the instruction writes $\sin c$ and $\cos c$ into two register components. The following rule combines sine and cosine instructions in the intermediate representation into a single `SCS` operation. The *wildcard pattern* (`*`-pattern) denotes that the two instructions need not be adjacent to each other. The curly braces in the search pattern indicate an *unordered* sequence: The instructions may be matched in any order.

```
rule scs {
  search:    { UnOp($$->opcode == OP_COS),
               *,
               UnOp($$->opcode == OP_SIN) }
  condition: { return $1->op == $3->op; }
  cost:      { return 1; }
  replace:   [ GPUUnOp(OP_SCS, SymReg ($1->tgt, $3->tgt),
                       $1->op) ]
}
```

Figure 1 illustrates the effect on the input and output instruction sequence after applying the rule `scs`. The figure demonstrates that the pattern matcher must not apply the rule in every case. For instance, if an instruction in-between the matched instructions modifies the operand or the target of the second matched instruction, the transformation will most likely modify the semantics of the input program and is thus invalid in general. Note that every rule with at least on wildcard pattern in its search pattern is subject to this negative side-effect. Thus, a special, semantics-preserving check is required.

The OORS language enables the developer to implement such a semantics-preserving check. The developer implements a *global implicit condition*, which decides whether the rule in question may be applied. It does so by checking for data dependencies between the matched instructions, which would prevent a reordering. This implicit condition is only checked for rules whose search pattern contains a wildcard pattern.

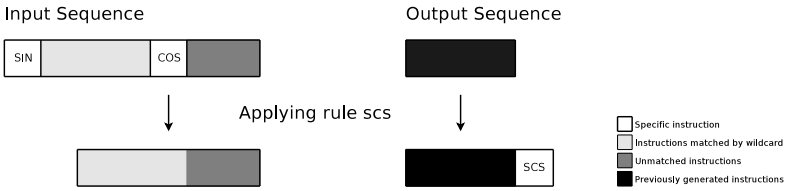

Fig. 1. Input and output instruction sequence before and after applying the rule `scs`.

Under certain circumstances, it might not be sufficient to generate the same sequence of instructions all the time. Some instructions might have to be translated into different instruction sequences depending on the type of their operators or similar side conditions. For this reason, the OORS language allows the developer to guard any sequence of instructions to generate via `if-then-else` statements. This enables the developer to integrate all possible alternatives into a single replace pattern, which keeps the pattern matcher specification readable. Example 4 shows the usage of guards within replace patterns.

Example 4 (Rule that generates different instruction sequences for the same instruction type). On GPU architectures, each register is a float vector comprising four components (called `r g b a`). In the source language, and thus in the intermediate instructions, operations work on scalars or on vectors with a length of at most 4; in general, the native arithmetic instructions also support such vectorial operations. However, some instructions operate only on scalar operands. Thus, special treatment is required if an operand of the correlative abstract instruction is of vector type. To resolve this problem, a sequence of the same scalar instructions has to be generated for each vector component. The rule `sin` generates code for the `SIN`-instruction, which computes the sine of its operand. When generating the corresponding GPU code, the rule has to make sure to select the

correct vector components (e. g., *cmp('r')* directs the instruction object to read from and write to the `r`-component).[1]

```
rule sin {
  search:  [ UnOp($$->opcode == OP_SIN) ]
  cost:    { return $1->target->components; }
  replace: [ if ($1->useComponent('r')) [
               GPUUnOp($1->opcode, $1->tgt, $1->op, cmp('r'))
             ],
             if ($1->useComponent('g')) [
               GPUUnOp($1->opcode, $1->tgt, $1->op, cmp('g'))
             ],
             if ($1->useComponent('b')) [
               GPUUnOp($1->opcode, $1->tgt, $1->op, cmp('b'))
             ],
             if ($1->useComponent('a')) [
               GPUUnOp($1->opcode, $1->tgt, $1->op, cmp('a'))
             ] ]
}
```

To further improve the maintainability and compositionality of the pattern matcher specifications, the OORS language introduces the notion of *intermediate replace patterns*. The developer specifies a replace pattern not on the instruction set of the target architecture, but on the intermediate instructions.[2] These intermediate instructions are then subject to the matching process as usual. The benefit of the intermediate instructions is to allow *compositional* specifications. For example, a vectorial `tan` instruction has to be implemented by sequences of native `sin` and `cos` instructions just like in Example 4, followed by a division; a much more complicated way than just specifying an *intermediate level* `sin-cos-div` sequence and letting the matcher then generate the component-wise native instructions from these intermediate instructions.

3.2 Rule Sets

An OORS pattern matcher specification comprises rules, which are organized into *profiles*. A profile represents a set of rules dedicated to a specific target architecture. During runtime, the host compiler (the compiler using the generated code-generator) selects the corresponding profile to be used for processing the input program(s). In this section, we first describe the matching process with respect to a single profile. Afterwards, we discuss the specifications mechanism concerning multiple profiles and their relationships.

Matching For a given profile, the pattern matcher tries to match the input instruction sequence against the search patterns until every input symbol has

[1] The components need not be consecutive.
[2] This kind of rule is specified by the keyword `intermediate` instead of `replace`.

been covered. In a *greedy* matching mode, the input is processed left-to-right, where always the rule with the lowest costs is selected. In case of a tie, the first specified rule is chosen. The pattern matcher employs backtracking in case the current match cannot be enlarged further while some input instructions remain unmatched.

In contrast to the greedy matching, an *optimal* matching mode investigates all possible matches to select the one with the globally optimal costs. If costs are not negative, we do not need to exhaustively explore the search space, but can prune the search space as soon as it can be determined that the current, incomplete match cannot outpace a previously found match.

Inheritance To keep an OORS pattern matcher specification readable and thus maintainable in the long run, the OORS language comprises rule and profile *inheritance* mechanisms. The developer can derive a new profile from existing ones. The new profile inherits all rules, may add new rules and may omit and modify inherited rules. Example 5 demonstrates the profile and rule inheritance mechanism.

Example 5 (Profile and rule inheritance). The source language features a dot-product operating on vector operands. The target architectures support the dot-product for vectors of three and four components. The newer G80 architecture supports also the dot-product for vectors of two components, whereas before it had to be realized using multiplication and addition. Thus, the NV40 profile comprises two distinct rules to cover all kinds of operands.

```
profile NV40 {
  rule dp2 {
    search:    [ BinOp($$->opcode == OP_DP) ]
    condition: { return $1->target->components == 2; }
    cost:      { return 2; }
    replace:   [ /* MUL, ADD */ ]
  }
  rule dp : extends dp2 {
    condition: { return $1->target->components > 2; }
    cost:      { return 1; }
    replace:   [ GPUBinOp(OP_DP, $1->tgt, $1->op1, $1->op2) ]
  }
}

profile G80 : extends NV40 {
  omit NV40::dp, NV40::dp2;
  rule dp : extends NV40::dp {
    condition: { return $1->target->components > 1; }
  }
}
```

The G80 profile is specified as an extension of the NV40 profile. So, it inherits per default all NV40 rules. The G80 profile omits the two NV40 rules `dp` and `dp2` and specifies a new, general rule as a modification of the inherited `dp`-rule.

3.3 Optimization

The pattern matching approach can also be used for code optimization. In contrast to the generational mode discussed in Section 3.2, there is only one sequence upon which the matching is performed. The pattern matcher searches for instances of the search patterns and literally replaces the matched instructions with the generated instructions. This process is repeated until no more rules can be applied, that is, until a fixpoint has been reached. Thus, there is no need for backtracking.

In this way, it is possible to represent low-level code optimizations in OORS, such as instruction rescheduling (see Example 6, Figure 2) or instruction merging (such as `MUL` and `ADD` into `MAD`).

Example 6 (TriCore instruction rescheduling optimization). On many recent architectures, the instruction order has a major influence on the execution time. For instance, the TriCore architecture [18] is only able to dispatch two instructions at once, if the first instruction will be executed in the arithmetic-logical unit (ALU) and the second instruction will be issued to the load-store unit (LSU). So, the following rule tries to pull a distant memory instruction behind an arithmetic-logical instruction. Although the implicit condition will verify whether possible side effects occur, the developer has to check manually whether it is safe to push the memory instruction in front of the second matched instruction (see the `condition` line; the absence of conflicts with the wildcard is guaranteed by the implicit condition). The cost function favors the match with the greatest distance between the ALU- and the LSU-instruction (for a wildcard pattern, the `$`-operator denotes the number of instructions the pattern has matched). Figure 2 shows this graphically.

```
rule reschedule {
  search:    [ Op($$->isIssuedTo(ALU)),
               Op(!$$->isIssuedTo(LSU)),
               *,
               Op($$->isIssuedTo(LSU)) ]
  condition: { return !$2->conflictsWith($4); }
  cost:      { return -$3; }
  replace:   [ $1, $4, $2 ]
}
```

4 Practice

The integration of OORS pattern matchers works in much the same way as for tools such as yacc or flex (see Figure 3). From the pattern matching specification, OORG generates a C^{++}-file implementing the matcher. This file has to be compiled and linked with the main compiler. The compiler selects at runtime the profile to be used in a particular compilation and calls the generated matcher for each basic block, passing the sequence of intermediate instructions and receiving

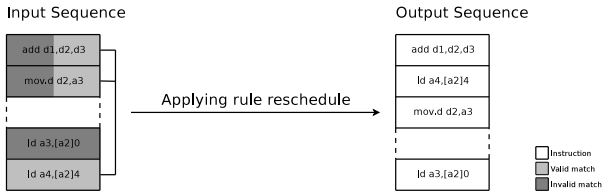

Fig. 2. Valid and invalid match of the `reschedule` rule. The first match (dark-gray) is invalid, because there is a definition-use dependency between `mov.d d2,a3` and `ld a3,[a2]0` (the first instruction reads from and the second instruction writes to register `a3`). The rule's condition prevents this match from being accepted.

the sequence of target instructions. To easily cater for minor variations of the desired rule set, the application program can switch on and off certain rules. For example, different sets of optimizations can be selected in this way.

The CGiS compiler [14, 15] can compile a common input program for GPUs and for SIMD CPUs. OORG-generated matchers are employed in three phases; the actual matchers used in a compilation are selected at runtime depending on the actual target. An early optimizer performs various transformations on intermediate code, which are needed for implementation on SIMD CPUs. A generator transforms the sequence of of intermediate instructions, which are common for GPUs and SIMD CPUs, into a target-specific instruction sequence. To this end, there are three hierarchies of profiles: For various generations of GPUs, SSE, and AltiVec. In a later phase, the GPU code is transformed by an OORG-generated peephole optimizer.

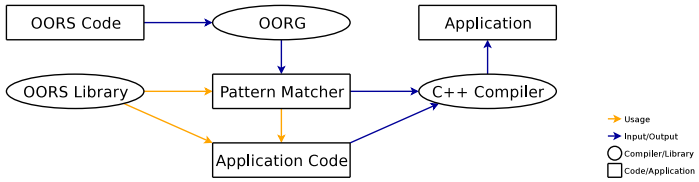

Fig. 3. Integration of OORS into an existing application.

We have implemented an NV40 code generation pattern matcher that comprises 43 rules, and a G80 code generation pattern matcher consisting of 50 rules. The G80 code generation profile inherits most NV40 rules, but replaces some NV40 rules with more specialized ones. The NV40 and the G80 profile of the code optimization pattern matcher comprise 12 rules. Both optimization profiles use the same rule set.

For both profiles, the CGiS compiler spends approximately 10% of the total compile time within the code generation and the code optimization pattern matcher. So, the influence of OORG-generated code generation and code optimization pattern matchers of the overall runtime is negligible.

5 Conclusion and Future Work

In this paper, we have presented the novel object-oriented rewrite system OORS with applications in code generation and code optimization. Using the presented OORS language, a developer is able to implement the code generation and code optimization phase of a compiler's back end in terms of pattern matchers. Retargetability is achieved by organizing rules into profiles, one for each supported hardware architecture. In contrast to other approaches, the OORS language features constructs, such as a rule and profile inheritance mechanism, that make the reuse of existing specification possible. Thus, an OORS pattern matcher specification is maintainable as well as easily retargetable in the long run.

We have additionally introduced the pattern matcher generator OORG that compiles an OORS specification into a C^{++} dynamically retargetable pattern matcher. By means of the CGiS compiler, we have demonstrated the usage of OORG-generated pattern matchers in a real world application. OORG is open source and available for download on our homepage.

The OORS language offers room for further improvements. Currently, OORS pattern matcher process only basic blocks. This restriction decreases the efficiency of certain optimizations, such as dead-code elimination. A dead-code elimination rule is currently not able to determine whether the target register is still live, if e. g., a register is written at the end of a basic block. Thus, we want to extend OORS such that matches over the whole control flow graph are possible. Apart from that, all instructions are assumed to be pushed upwards past wildcard patterns (remember Example 6, where a load-store instruction is pushed upwards). In some cases however, a developer might want to push instructions the other way around. To further improve the expressiveness of OORS, we thus want to introduce a mechanism that indicates the *direction* of a rule. Finally, we plan to improve the performance of the OORG-generated pattern matchers. The current implementation generates pattern matchers that match each rule one after another. This matching method becomes inefficient if the search patterns of two (or more) rules share the same prefix. In such a case, the OORG-generated pattern matcher would match that prefix multiple times. So, to overcome this drawback, we want to improve OORG, such that the generator produces pattern matchers that match all rules in parallel.

Acknowledgements

We thank the anonymous reviewers for their encouraging and helpful comments about this paper. Due to lack of space however, we were not able to add more content regarding further details about OORS or comparisons with other, similar approaches.

References

[1] M. Alt, U. Aßmann, and H. van Someren. Cosy Compiler Phase Embedding with the CoSy Compiler Model. In *5th International Conference on Compiler Construction*, volume 786 of *LNCS*, pages 278–293. Springer-Verlag, 1994.

[2] M. Breternitz Jr., H. Hum, and S. Kumar. Compilation, Architectural Support, and Evaluation of SIMD Graphics Pipeline Programs on a General-Purpose CPU. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 135, Washington, DC, USA, 2003. IEEE Computer Society.

[3] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[4] Quick C-- compiler. `http://cminusminus.org/qc--.html`.

[5] J. W. Davidson and C. W. Fraser. Register Allocation and Exhaustive Peephole Optimization. *Software – Practise and Experience*, 14(9):857–865, September 1984.

[6] J. Dias and N. Ramsey. Converting intermediate code to assembly code using declarative machine descriptions. In *CC*, pages 217–231, 2006.

[7] H. Emmelmann, F.-W. Schröer, and R. Landwehr. BEG – A Generator for Efficient Back Ends. In *PLDI*, pages 227–237, 1989.

[8] S. Farfeleder, A. Krall, E. Steiner, and F. Brandner. Effective compiler generation by architecture description. In *LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems*, pages 145–152, New York, NY, USA, 2006. ACM Press.

[9] C. Ferdinand, H. Seidl, and R. Wilhelm. Tree Automata for Code Selection. *Acta Informatica*, 31(9):741–760, 1994.

[10] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[11] C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG – Fast Optimal Instruction Selection and Tree Parsing. *SIGPLAN Notices*, 27(4):68–76, 1992.

[12] G. Gebhard and P. Lucas. OORS: An Object-Oriented Rewrite System with Applications in Retargetable Code Generation and Optimization. Technical report, University of the Saarland, 2007. To appear.

[13] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated Soundness Proofs for Dataflow Analyses and Transformations via Local Rules. *SIGPLAN Notices*, 40(1):364–377, 2005.

[14] P. Lucas, N. Fritz, and R. Wilhelm. The CGiS compiler—a tool demonstration. In A. Mycroft and A. Zeller, editors, *Proceedings of the 15th International Conference on Compiler Construction (CC)*, volume 3923 of *LNCS*, pages 105–108. Springer-Verlag, 2006.

[15] P. Lucas, N. Fritz, and R. Wilhelm. The Development of the Data-Parallel GPU Programming Language CGiS. In *International Conference on Computational Science (4)*, volume 3994 of *Lecture Notes in Computer Science*, pages 200–203. Springer Verlag, 2006.

[16] NVIDIA. Nv_gpu_program4. OpenGL Extension 322, 2007.

[17] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[18] TriCore microcontroller. `http://www.infineon.com/tricore/`.