Graph Rewriting for Hardware Dependent Program Optimizations

Andreas Schösser and Rubino Geiß

Universität Karlsruhe (TH), 76131 Karlsruhe, Germany
{andi|rubino}@ipd.info.uni-karlsruhe.de

Abstract. We present a compiler internal program optimization that uses graph rewriting. This optimization enables the compiler to automatically use rich instructions (such as SIMD instructions) provided by modern CPUs and is transparent to the user of the compiler. New instructions can be introduced easily by specifying their behaviour in a high-level programming language. The optimization is integrated into an existing compiler, gaining high speedup.

1 Introduction

Current programming languages don't pay much attention to *rich instructions* provided by recent CPUs. By *rich instruction* we mean a small program implemented in hardware, consisting of several conventional instructions and capable of operating on multiple data in parallel. For example, SIMD[1] instructions fall in this category. Rich instructions are applied to benefit from shorter execution time compared to executing conventional instructions. Programmers have different options to take advantage of rich instructions:

Using assembly language. This option can quickly lead to a huge programming effort and maintenance overhead.

Using compiler specific intrinsics. Requires adaptation of a program when changing the target architecture or the compiler. Moreover, existing programs can only be optimized if their source code is rewritten manually.

Using a compiler internal optimization. Transparent to the programmer; the compiler decides automatically when to use a rich instruction instead of several basic instructions, depending on the target architecture.

Assembly language and intrinsics are not applicable since we want to keep the source code portable. Most compilers currently available don't fully support optimizations mentioned above. Some compilers use a so-called *Vectorizer* to vectorize loops but don't optimize programs outside loops and fail for rich instructions which are more complex than pure vector instructions.

To overcome these limitations we present a new approach to implement such an optimization. Optimizations are usually done on a compiler internal intermediate program representation (IR). Modern IRs are graph based, consisting

[1] Single Instruction Multiple Data

of nodes representing operations and edges representing data and control flow. Since an IR needs to be hardware independent, it does not initially contain node types for hardware specific rich instructions. Instead, we can find rich instructions as *subgraphs* of an IR graph. These subgraphs are composed of several basic instructions. To perform an optimization, we transform the original graph by replacing these subgraphs by a single corresponding, hardware specific node[2].

Figure 1 shows a small example of how such an optimizing transformation looks like. The graph shown on the left—executing two *Load* operations on consecutive storage locations starting at a base address *Base*—is replaced by the graph shown on the right, which executes both *Load* operations at once by using a rich instruction named *VectorLoad*. *VectorLoad* delivers the same results as the graph we replaced. To distinguish the two partial results of the *VectorLoad* instruction, we use edge types *Result 1* and *Result 2*. Note that data dependency edges instead of data flow edges were used in this example. That is, to get the execution order of the statements you have to read the dependency edges backwards.

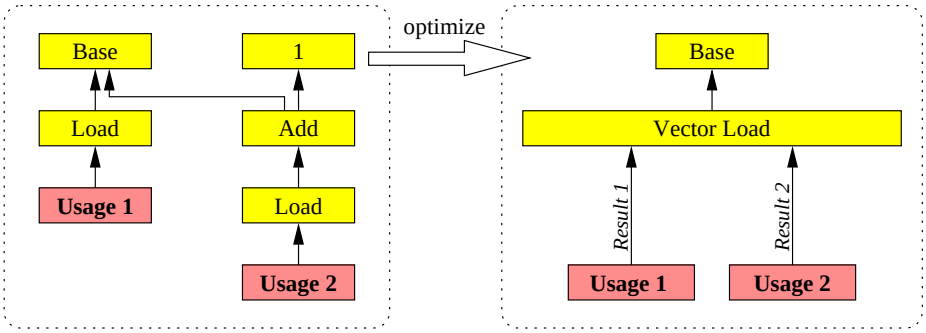


Fig. 1. Optimizing transformation of an IR graph

This paper presents new techniques to perform such a transformation:

- In the field of compiler construction it's common to do graph transformations manually. Though, finding a *pattern graph* in a host graph and replacing it by a *replacement graph* are tasks that can be delegated to a *graph rewrite system* (GRS) (Section 2). Doing this allows us to specify graph transformations in an abstract way.
- Up to now, *pattern graphs* had to be specified manually. This method is very time consuming and error-prone (Section 2). Our idea is to *generate* the pattern graphs from a specification that describes the *behaviour* of a rich instruction in a standard programming language (Section 3).

[2] Therefore, we require that the IR is extendable and especially enables the introduction of hardware specific nodes after high level optimizations are completed.

– We show how to automatically generate and apply graph rewrite rules to perform program optimization (Section 4).
– We provide benchmark results to show the benefit gained by our optimization and that it is performed by the GRS in admissible time (Section 6).

2 The Problem

Our first problem to solve is choosing a graph rewrite system suitable for our needs. We use *GrGen* [1, 2] which is a well-known and fast graph rewrite tool. It features an extensive specification language and can operate directly on our compiler's IR [3]. We don't want to search and replace patterns manually because pattern graphs can grow huge in our case. This is because of the complexity of rich instructions on the one hand and the complexity of the IR on the other hand.

2.1 The Complexity of Rich Instructions

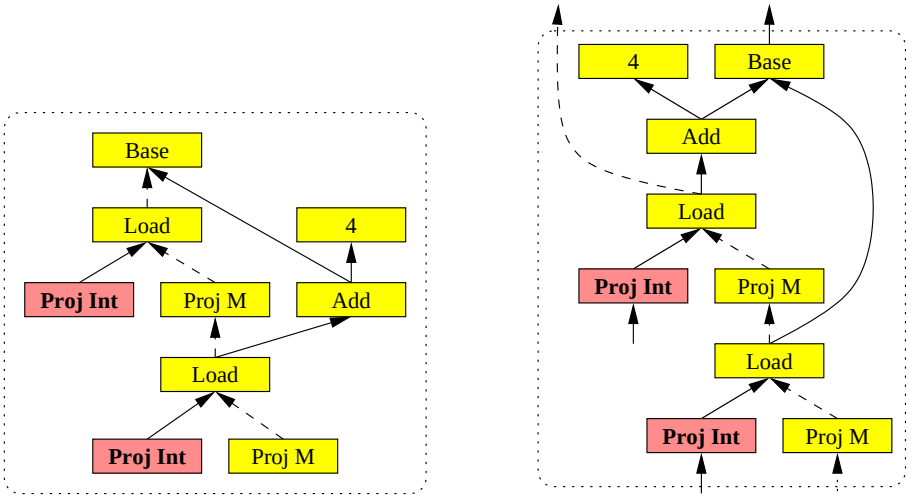
We already presented a rather simple *VectorLoad* instruction, but also more complex instructions, e.g. incorporating forked control flow, are possible. For example, the following C code calculates the *Sum of Absolute Differences (SAD)* of two vectors and could be replaced by a rich instruction named *psadbw* taken from the *Intel SSE2* instruction set [4].

```
unsigned char a[16], b[16];
int result = 0, i;
...
for(i = 0; i < 16; i++) {
   if(a[i] > b[i])
      result += a[i] - b[i];
   else
      result += b[i] - a[i];
}
```

When applying rich instructions, we have to take into account that they often handle *vectors* instead of *scalar values* and can operate on special *vector register* sets.

2.2 The Complexity of an IR

The complexity of rich instructions together with the complexity of an IR makes it very difficult to integrate new rich instructions manually, even for experts. For example, one of the main problems when creating pattern graphs is program variation, i.e. pattern graph and host graph may differ even though both have equivalent semantics. Since the problem of proofing the equivalence of two programs is undecidable in general, it's not possible to find *every* subgraph having the same semantics as the pattern graph. Yet, we can try to find as many subgraphs as possible by applying special techniques like *normalization* and *creating*



(a) Pattern graph of a *VectorLoad* instruction

(b) Host graph containing a *VectorLoad* subgraph

Fig. 2. Difference between pattern graph and host graph

variants. Figure 2(a) shows an example pattern graph for a *VectorLoad* instruction which differs significantly from the host graph given in figure Figure 2(b). Both graphs are variants of the code fragment

```
int a, b, c[2];
...
a = c[0];
b = c[1];
...
```

transformed to our IR called Firm [5]. To help you understand this example, we have to provide you with some technical details about Firm:

Firm is a graph based IR which satisfies the SSA[3] property [6, 7]. Most operations in Firm are self-explanatory, yet there are some node types to be explained more closely. Nodes of type *Proj* are used to simulate *edge types*, a concept with doesn't exist in Firm. Hence, they won't produce any assembler code. Edges of type *memory* (denoted as *memory edge* in the following) are used to serialize memory operations in order of appearance in the source code. They are marked with a *Proj M* node and drawn as a dotted line for clarity. Edges of type *integer*, marked with a *Proj Int* node, represent an integer result. In Firm, *Proj* nodes are omitted if the edge type is non-ambiguous, that is, if the edge type is implicitly given by the result type of the edge's target node.

The difference between the pattern graph in Figure 2(a) and the host graph in Figure 2(b) is that the *Load* operations are serialized in a different order

[3] Single Static Assignment

by memory edges. Other differences not shown here may occur in arithmetic expressions, which is important especially in the context of address calculation. For example, the simple expression $a + b + c$ can be scheduled as $(a + b) + c$ or $a + (b + c)$. Note that it depends on the programming language whether re-ordering of arithmetic expressions is allowed.

In general, the patterns are much more complex, including several basic blocks and forked control flow (not modelled here for conciseness). By creating pattern graphs manually, it's easy to make mistakes due to the complexity of the IR, even for experts. Therefore, introducing new rich instructions to the compiler is very time consuming this way. Moreover, changes made to the IR require the patterns to be rewritten. We're heading for a solution to generate graph rewrite rules automatically.

2.3 Inserting Rich Instructions

Further questions arise when it comes to create the replacement graph which is used to insert the new rich instruction. At first, we have to decide where to schedule the rich instruction. Secondly, it's important to connect the new instruction to the right operands. Moreover, we have to investigate a way to make the result of the rich instruction public so that up-following instructions are able to use it. This is not trivial because rich instructions may save their results in vector registers, which conventional instructions cannot access. We therefore have to find a way to transport data to those vector registers and back.

After graph rewrite rules have been created, we need a method to decide which rules we shall apply. This is especially interesting when different subgraphs to be replaced overlap. In some cases, we're not allowed to replace a subgraph due to constraints of the IR in order to keep the semantics of the rewritten program exactly as it was before the rewrite. Hence, we have to analyze the host graph carefully before replacing.

A solution to these problems is presented in the following sections.

3 Using GrGen for Program Optimization

As described in Section 2, creating pattern graphs by hand is an arduous task, especially for complex rich instructions. Our solution is to specify *the behaviour* of rich instructions in a standard programming language and generate the graph rewrite rules *automatically*. In the following, we present an instruction specification language based on the programming language C and show how to integrate such an optimization into an existing compiler. The optimization is divided into two steps described in the following sections.

3.1 Generating Graph Rewrite Rules

The first step is to generate graph rewrite rules to be used for optimization. The pattern and replacement graphs are derived from specifications of the behaviour

Listing 1. Example specification of a rich instruction

```
1  void VectorAdd(void)
2  {
3     /* Definition part */
4     double *a = Operand_0("vector", "memory",   "gp");
5     double *b = Operand_1("vector", "register", "xmm");
6     double *res =  Result("vector", "register", "in_r1");
7     Emit("addpd %S0, %S1");
8
9     /* Behaviour of the instruction */
10    res[0] = a[0] + b[0];
11    res[1] = a[1] + b[1];
12 }
```

of a rich instruction stated in the programming language C. Thus, the instruction specifications can easily be drawn from pseudo code descriptions of CPU instructions used in reference manuals (cf. [4, 8]).

Instruction Specification. Listing 1 gives an example of how such a specification looks like: It consists of one or more functions, each function describing the behaviour of a rich instruction. In this example, the instruction `VectorAdd` is specified. Each function consists of a *definition part* and a *behaviour part*.

In the definition part, a call of the function `Operand_n` or `Result` defines a variable the rich instruction operates on. `Operand_n` means that the variable represents the nth operand of the instruction, `Result` means that the variable represents it's result. Each variable has additional attributes, which are set by passing parameters to those functions. These attributes specify the kind of data the variable represents (`vector` or `scalar`), the location of the data (`register` or `memory`) and the name of the register class in which the data is passed. Beyond variable definition, the assembler code to emit has to be specified here, using the function `Emit(char *assembler_template)`. Our example uses the *Intel SSE3* instruction set and emits the instruction `addpd` which adds two vectors of two double precision floating point components. The wildcards `%S0` and `%S1` stand for source register 0 and 1, in which the operands 0 and 1 are passed. The compiler backend replaces these wildcards by the actual registers it allocated.

The behaviour part describes the exact behaviour of the rich instruction in plain C, using the variables defined in the definition part. The instruction presented in this example performs a vector addition of two vectors given by the variables `a` and `b`, and writes the result to the location represented by the variable `res`.

Note that the specification uses plain C syntax for the definition and behaviour part. We just added some extra semantics. The elements used in this example can be used to specify a wide range of rich instructions. Yet, the specification language can be enhanced easily (see sections 4.4 and 4.7).

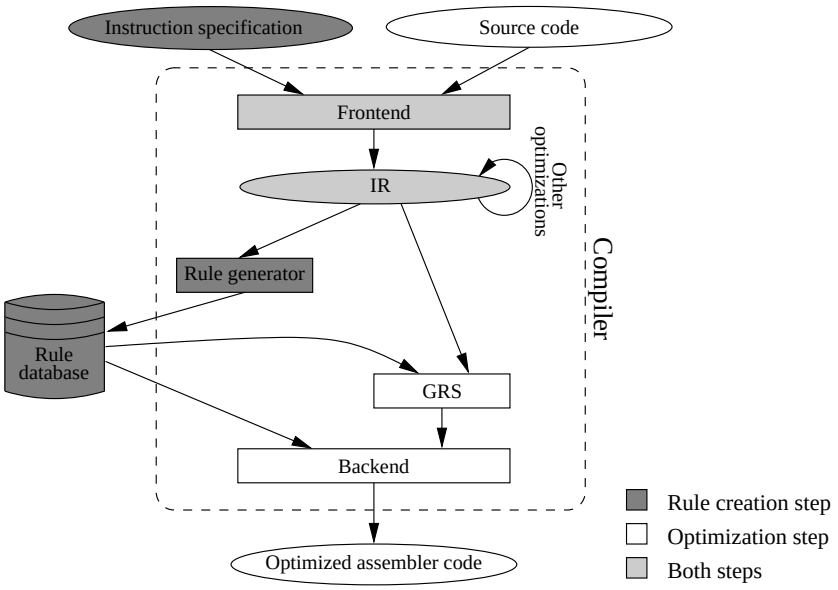


Fig. 3. Compiler integration

Integration. Figure 3 shows how we integrate the rule creation step into an existing compiler (marked dark and light grey): We use the unmodified *compiler frontend* to transform the *instruction specification* to an *IR* graph (called *initial pattern*). This is possible because the specification consists of plain C code. The initial pattern is analyzed and then transformed to the pattern and replacement graph by the *rule generator*. The generated graph rewrite rules are saved in an appropriate format in a *rule database* to be used by the *GRS* in the optimization step. In addition, we need to tell the compiler backend about the newly introduced instruction. Therefore, we also generate annotations for the backend in order to be able to produce the right assembler code. All information needed is found in the initial pattern.

This rule creation step has to be performed only once when new rich instructions are introduced. After the *rule database* has been filled, it can be used to perform the optimization step.

3.2 Performing the Optimization

In the second step, we use the rules saved during the first step to perform the actual optimization (marked white and light grey in Figure 3). First we use the compiler frontend to transform the *source code* to an *IR* graph, on which we can perform pattern matching using the *GRS*. A subgraph found, representing a rich instruction, is called a *match*. Beyond pattern matching and replacement, there are further tasks to perform during the optimization step. Because we generate the pattern graph as general as possible to avoid being too restrictive, we have to

test whether we're really allowed to replace a match before actually replacing it. During the rewrite, a new node representing the rich instruction is introduced. We have to schedule this node correctly, e.g. put it in the right basic block and serialize it correctly with regard to its memory dependency. We will discuss this more closely in Section 4.

4 Implementation

In this section, we present an outline of our implementation.

4.1 Preliminary Transformations

Before our optimization is launched, other graph transformations are performed in order to reduce program variation. These transformations are standard optimizations common in modern compilers, but also special normalizing transformations are applied:

Lowering. Our optimization is performed after the FIRM graph has been *lowered*. That means high-level constructs to access array components have been replaced by pointer arithmetic. This way, we can handle direct array access as well as array access by pointer arithmetic in the source program.

Removing critical edges. Removing critical edges saves us from doing normalization in special cases when control flow is forked.

Loop unrolling. We might not find many rich instructions in ordinary programs since the access of vector components is hidden behind different loop iterations. Unrolling inner loops n times while n being the maximal vector size might reveal those hidden instructions.

Load-Store optimization. An optimization which uses information returned by an alias analysis to eliminate unnecessary *Load* operations and deserializes memory accessing operations if possible. Therefore, a good alias analysis is essential for our needs. We use a so-called *memory disambiguator* built into our compiler [9].

Special normalizations. For example, arithmetic expressions used for address calculation are brought to a consistent form. Hofmann presented a way to normalize generic arithmetic expressions [10].

4.2 Matching

In this paragraph, we describe the pattern creation and pattern matching process more closely. We start with the initial pattern delivered by the compiler frontend. The initial pattern contains the operations of the specification's definition and behaviour part, whereas, for the pattern graph only the behaviour part is needed. We designed the instruction specification language in the way that all information the user specified can be regained from the initial pattern. To generate the pattern graph, we first extract all important information the user

specified in the definition part and then clip the initial pattern by the nodes not needed any more.

The pattern graph now consists of the nodes representing the behaviour part. The names of the variables defined in the instruction specification are not important any more, because FIRM graphs satisfy the SSA property. That means each data flow edge represents a *value* and by matching the dataflow edges exactly we make sure that matches found in the host graph have identical behaviour as the pattern graph.

Matching the exact dataflow must not be confused with matching the statement-level control flow. We do not match the statement-level control flow because only the control flow between basic blocks is represented in FIRM. All possible control flows inside a basic block are implicitly given by the dataflow. This makes the pattern graph more general with regard to the source program to be optimized.

In section 2 we stated that we want to deal with program variation shown in Figure 2. The problem is the serialization of memory operations by memory edges. Our solution is not to match the memory dependency at all. Instead, we check the memory dependencies for consistency after finding a match. To do so, we take advantage of a feature of *GrGen* to be able to insert processing between matching and rewrite.

The initial pattern may contain *Load* and *Store* operations which represent an access to a vector register instead to a memory location, depending on the user specification. To insert this information explicitly into the graph, we introduce a new node type *VProj* representing a specific component of a vector register. The component number is given by a node attribute.

In general, *edge positions* are important in FIRM. That's because nodes represent operations and the operand order is very important for example for a *Sub* operation. Hence, we check edge position numbers exactly except for commutative operations like *Add* or *Mul* of integer type.

4.3 Replacement

At first glance, it seems obvious that inserting a new rich instruction after finding a match allows us to delete all the basic operations contained in that match. On closer examination we recognize that executing the basic operations might produce intermediate results not produced by the rich instruction. If operations outside the match use those intermediate results, the code that calculates them must not be deleted. Therefore, the nodes contained in the pattern graph are also included in the replacement graph, except for the end results also produced by the rich operation. Unnecessary basic instructions are deleted in a later step, when it turns out that their result is not used. This is done by standard compiler optimizations like the *Dead Node Elimination*, the *Control Flow Optimization* and *Load-Store Optimization* [11].

Figure 4 shows how the replacement works in general. The left hand side represents a match we found. It uses n operands—*Op 1* to *Op n*—and a set of basic *operations* to calculate a result vector of m components—*Res 1* to *Res*

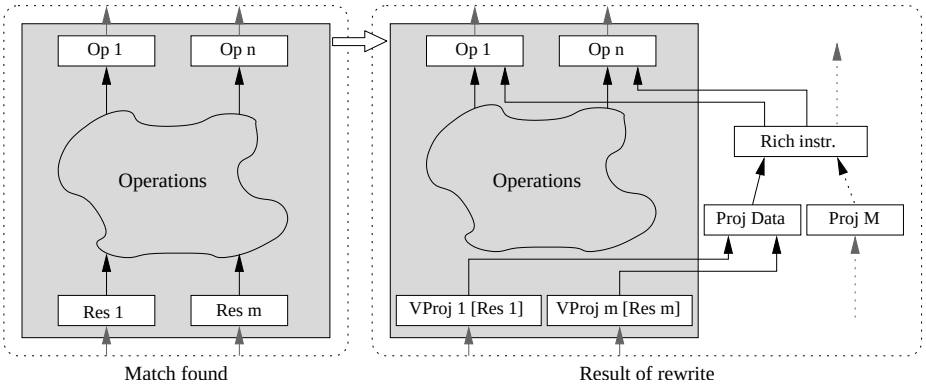


Fig. 4. Match found (left) and result graph (right).

m. Now we want this match to be rewritten so that the corresponding new rich instruction is inserted and the program uses the result of the rich instruction.

This is shown on the right hand side which corresponds to the host graph after the rewrite. The rewrite step inserts a new node representing the rich instruction. The operands of the new instruction are already contained in the found match, so the rewrite step also connects the rich instructions to it's operands. The number of usages is not known at pattern creation time. In order to connect the results of the rich instruction, we would have to reroute an unknown number of edges, which is hard to express with one single graph rewrite rule. To avoid this problem, we *retype* the previous result nodes to *VProj k* nodes, which indicate that this node represents the $k'th$ component of the result vector. Also, the retyped nodes are disconnected from their previous operands and connected with the result of the rich instruction. This way each node that previously used an end result calculated by basic instructions now uses an end result calculated by the rich instruction. A *VProj* node indicates, that the result of the rich instruction is passed in a vector register and if basic instructions want to use the result, some kind of code which transfers the results to general purpose registers has to be inserted.

After rewriting is finished, we have to perform further transformations which can not be expressed through the graph rewrite rule. At first, the new rich instruction has to be inserted into a certain basic block. We insert the rich instruction in the basic block which dominates all other basic blocks contained in the match because we want to make the results of the rich instruction available as soon as possible. This can not be done by the GRS since we can't know the exact basic block layout of the host graph at pattern creation time. It's important that the operands of the rich instruction are available *before* the rich instruction is scheduled. Otherwise a *deadlock* occurs which means that an operation was scheduled without its operands being available. We prevent deadlocks by

analysing the host graph before rewriting and only apply a rule if it's safe to rewrite.

Secondly, we have to insert the rich instruction into the memory dependency chain. Again, to make the results of the rich instruction available early, we place it before any memory operation contained in the match. Please note, that this method prevents deadlocks but also introduces further issues: If the rich operation writes to a memory location it has already read by a *Load* node, then this *Load* is not allowed to have usages outside the match. The node that uses the *Load*'s result would receive an invalid value. To recognize this and similar cases, detailed analysis is necessary.

4.4 Priorities

Introducing *VProj* nodes makes graph rewrite rules dependent on each other. Consider, for example, rules for a *VectorLoad* and a *VectorStore* instruction. We assume that the *VectorStore* expects data located in a vector register, represented by several *VProj* nodes in the Firm graph. The *VProj* nodes are not initially in the Firm graph, they are produced by the *VectorLoad* graph rewrite rule. That's why the *VectorLoad* rule has to be applied before the *VectorStore* rule. To specify that, we introduce *priorities*. The user can assign a priority class to each rule and put rules depending on each other into ascending priority classes. Rules not depending on each other may remain in the same priority class. The syntax to assign an instruction to a priority class—e.g. 5—is

```
Priority(5);
```

4.5 Variants

As a result of loop unrolling, the values of vector components are often dependent on one or more induction variables which do not occur in the specification of the rich instruction. In this case, one or more additional summands have to be added to all parts of the pattern which calculate a memory address. We create variants for each additional summand. This way, we can even handle access to vector components which are located in multi-dimensional arrays.

4.6 Replacement Strategies

Having several graph rewrite rules at stock, the optimization has to apply these rules automatically. We want to apply the rules resulting in the maximal saving of costs according to a cost model. The saving of costs for each replacement can be pre-calculated by taking the costs of the operations to replace and the costs of the rich instruction into account. Dynamic costs occur when matched nodes can not be deleted because they deliver an intermediate result (see Section 4.3). The most problematic situation we have to deal with is overlapping matches.

Unfortunately, we can not locally decide which match to choose for replacement because we don't know which subsequent matches will follow and in which

saving of costs this will result. One approach would be to replace one of the overlapping patterns and determine its cumulative costs and then use a *roll-back* function to re-establish the graph to try out the second pattern. Seeing how subsequent matches depend on *VProj* nodes, our approach is not to replace overlapping matches at first but only insert all the corresponding *VProj* nodes into the graph in parallel. Thus, subsequent matches can also be found in parallel, making a cost intensive rollback function unnecessary. This approach completely separates the pattern matching step from the rewrite step. All matches that can be found in an IR graph and their savings of costs are represented in a so-called *search tree*. A walk over the search tree selects all matches to be rewritten, thus resulting in an optimal solution according to our cost model. One problem when applying this explorative approach is that search trees can grow big. To reduce the size of the search tree, it's possible to build a search tree for each priority class only (see Section 4.4). This works fine for all our test programs and covers most, but not all, dependencies between rich instructions. Therefore, our current research is to use a heuristic PBQP[4]-solver for rule selection [12, 13].

4.7 Clean-Up Operations

It might occur that there are still *VProj* nodes in the graph but no subsequent match can be found. The compiler backend however, cannot select code for *VProj* nodes. A solution to the problem is to perform an undo-operation which rolls back the last replacement(s) until no *VProj* nodes are present in the graph any more.

The other possibility is to convert the *VProj* nodes. This is not trivial because the value represented by a *VProj* node has to be extracted from a vector register and copied to a general purpose (GP) register. Some processor architectures (like SSE2) don't have dedicated instructions to do so[5].

This means that two instructions have to be performed: shifting and copying. This can easily be done using our specification language by declaring both instructions in the *Emit* statement separated by a line break. This way, a *virtual instruction node* is created representing both operations:

```
Emit(".␣psrldq␣$8,␣%S0\n.␣movq␣%S0,␣%D0");
```

This virtual instruction not only destroys the value contained in the destination register, but also the one contained in the source register. Therefore, the user can specify

```
Destroys(Operand);
```

to indicate that an operation also destroys the register which contains `Operand` besides the register where the result is stored.

[4] Partitioned Binary Quadratic Problem

[5] Only extracting single-word integer values is possible with SSE2 by using the `pextrw` instruction. A `pextrd` instruction is planned to be introduced with SSE4.

5 Related Work

5.1 Algorithm Recognition

Metzger and Wen extensively present an approach to recognize algorithms in the so-called *computational kernel* of a program and replace them by a call to an optimized library function, thus saving execution time [14]. The optimization is carried out on an IR that has tree shape and which was used in the *Convex Application Compiler* [15]. Data structures used are the *control tree* containing the statements and the control flow. *Expression trees* are used to represent expressions and the *i-val tree* representing the dependencies of induction values of loops. Also the statement-level data flow graph is computed. To speed up pattern matching the control tree, expression trees and the i-val tree are converted into a *canonical form*. This is especially useful for handling commutative operators. The reordering of the tree is based on an encoding of nodes of the control tree. Metzger and Wen also explain how to extract subprograms from the computational kernel and select the best *feasible replacement* to gain maximal benefit.

We share the same idea that a database of pre-created patterns has to be maintained and even that these patterns should be created by specifying an algorithm in a high-level language. This makes it easy for the end-user to add new patterns and optimizations can be performed without modifying the source code. Reducing program variation to keep the number of patterns to maintain small using standard compiler optimizations amongst others is an idea found in both approaches. We also seek a (good or even the best) selection of replacements for the patterns we found in order to accelerate the program.

We differ in the form of IR we use: A graph-based IR with integrated data and control flow, instead of a tree based IR. Metzger and Wen claim to find and replace complete algorithms including loops while we want to find DAGs (direct acyclic graphs) representing the behaviour of rich instructions. They have to extract subprograms for comparison by reordering the statements in the control flow tree as allowed by the dataflow. We don't have this problem, because we don't consider the statement-level control flow. Our optimization is more back-end oriented since we don't replace patterns by function calls but by hardware dependent assembler instructions. Hence we have to deal with hardware specific features like register classes when specifying new patterns. We use a modern GRS to match patterns instead of transforming IR programs to a canonical form and comparing patterns node by node. Finally, Metzger and Wen only implemented the normalization process and did not implement the pattern matching process.

5.2 Previous Implementations

Hofmann implemented an early version of the optimization presented in this article [10], which was mainly a proof of concept of how automatic pattern creation works. This first implementation wasn't able to optimize realistic programs automatically, and hence no serious run-time tests were possible.

6 Benchmarks

We have tested our implementation on a Pentium 4 (Prescott), 3.2 GHz, which features the *Intel SSE3* instruction set. The system has 2 GB main memory and runs Suse Linux 9.3.

Our test program implements a *block-matching* algorithm to perform *motion estimation* used for MPEG compression in video codecs [16]. The block-matching algorithm operates on a frame of 256x256 bytes size and is performed 100 times.

Listing 2. Motion estimation example

```
unsigned int sad(int test_blockx, int test_blocky, int *best_block_x,
                 int *best_block_y, unsigned char frame[256][256])
{
   int i, x, y, blocky, blockx;
   unsigned tmp_diff, min_diff = 0xFFFFFFFF;

   // Iterate over whole frame; x,y=coords of current block
   for(x = 1; x < 256 - 16; x++)
      for(y = 0; y < 256 - 16; y++) {
         tmp_diff = 0;
         // Compare current block with reference block
         for(blocky = 0; blocky < 16; blocky++) {
            for(blockx = 0; blockx < 16; blockx++)
               if(frame[blocky][blockx] > frame[blocky + y][blockx])
                  tmp_diff += (frame[blocky][blockx] - frame[blocky + y][blockx]);
               else
                  tmp_diff += (frame[blocky + y][blockx] - frame[blocky][blockx]);
         }

         // Check if the current block is least different
         if(min_diff > tmp_diff) {
            min_diff = tmp_diff;
            *best_block_x = x;
            *best_block_y = y;
         }
      }
   }
   return(min_diff);
}
```

The changes made to the host graph during optimization are presented in Table 1(a). The patterns found, and the assembler instructions applied, are shown in Table 1(b). Compared to compiling the program with conventional optimizations, we gain a maximal speedup of 32.26. Compared to programs compiled by the Intel C compiler, which also does hardware-specific optimizations, we still gain a speedup of 9.52. The whole optimization process took 1.2 seconds. *GrGen* spent only 40ms for matching 4480 nodes and 17853 edges and rewriting 4 matches. The rest of the time was spent for additional analysis and node elimination.

Table 1. Benchmark results.

(a) Execution time and graph statistics

	standard opt.	**rich instructions opt.**	**factor**
# nodes	2392	680	3.52
# edges	5324	1480	3.60
running time	13.55 s	420 ms	Speedup: 32.26

(b) Rich instructions applied

pattern	**applied**	**instructions**	**# nodes**	**# edges**
VectorLoad_16b_v1	1x	lddqu	82	96
VectorLoad_16b_v3	1x	lddqu	84	128
SAD_16b	1x	psadbw pshudf paddd	289	461
Component_0Iu	1x	movd	3	2

7 Conclusion

We have presented a novel optimization framework to speed up programs by using rich instructions. Our optimization works on the compiler internal IR and uses a GRS to find patterns representing rich instructions and automatically replaces them by a corresponding rich assembler instruction. The rules for the GRS are created automatically using a specification of the rich instruction's behaviour, based on the programming language C. Rules are selected automatically to receive the most efficient program according to a cost model. The optimization is integrated into an existing compiler.

The advantage is that source programs can be optimized without modification. Therefore, a source program stays portable and changing the compiler or the target architecture is still possible. In addition, the user can easily introduce new patterns for rich instructions without being a compiler engineer, because he can specify graph rewrite rules indirectly in a familiar language.

Validating our implementation with different test programs covering the fields video processing, sound processing and numerical calculations, showed that it outperforms traditional compilers, gaining speedup ranging from 1.05 to 32. Therefore, there is potential to optimize programs using rich instructions. The pattern matching core of the optimization, performed by the graph rewrite system *GrGen*, takes only milliseconds.

We haven't addressed rich instructions which require data to be specially aligned in memory, so far. When using such instructions, we have to test whether the data they access is aligned [17]. Additionally, we also need a good alias analysis. There are ways to support the alias analysis manually. E.g. the programmer could use the C qualifier `restrict` (C99) to indicate that a pointer has no alias. We're using an explorative algorithm to control rule application but are also researching on how a PBQP-solver could solve the problem more efficiently.

Acknowledgments. We thank all co-workers and students at the *IPD, Universität Karlsruhe* for their support and proof-reading, especially Christian Würdig, Christoph Hermann Mallon, Edgar Jakumeit, Gernot Veit Batz, Matthias Braun, Michael Beck, Moritz Kroll, and the anonymous reviewers. Also we thank Prof. Dr. Gerhard Goos for the generous support he provides at his chair.

References

1. Geiß, R., Kroll, M.: GrGen.NET: A Fast, Expressive, and General Purpose Graph Rewrite Tool. In: this volume. (2007)
2. Geiß, R., Batz, G.V., Grund, D., Hack, S., Szalkowski, A.: GrGen: A Fast SPO-Based Graph Rewriting Tool. In Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G., eds.: ICGT 2006. Volume 4178 of LNCS., Springer (2006) 383–397
3. Batz, G.V.: Graphersetzung für eine Zwischendarstellung im Übersetzerbau. Master's thesis, Universität Karlsruhe (2005)
4. Intel Corporation O. Box 5937, Denver, CO 80217-9808: Intel 64 and IA-32 Architectures Software Developer's Manual – Instruction Set Reference. (2007)
5. Trapp, M., Lindenmaier, G., Boesler, B.: Documentation of the Intermediate Representation FIRM. Technical Report 1999-14, Universität Karlsruhe, Fakultät für Informatik (1999)
6. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. ACM Trans. Program. Lang. Syst. **13**(4) (1991) 451–490
7. Cooper, K.D., Torczon, L.: Engineering a Compiler. Morgan Kaufmann Publishers Inc. (2004)
8. Motorola Phoenix, AZ, USA: AltiVec Technology Programming Environments Manual. (1998)
9. Ghiya, R., Lavery, D., Sehr, D.: On the Importance of Points-To Analysis and Other Memory Disambiguation Methods for C Programs. Proceedings of the ACM SIGPLAN 2001 PLDI (2001) 47–58
10. Hofmann, E.: Regelerzeugung zur maschinenabhängigen Codeoptimierung. Master's thesis, Universität Karlsruhe (2004)
11. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986)
12. Eckstein, E., König, O., Scholz, B.: Code Instruction Selection Based on SSA-Graphs. In Krall, A., ed.: SCOPES 2003. Volume 2826 of LNCS., Springer (2003) 49–65
13. Jakschitsch, H.: Befehlsauswahl auf SSA-Graphen. Master's thesis, Universität Karlsruhe (2004)
14. Metzger, R., Wen, Z.: Automatic Algorithm Recognition and Replacement: A new Approach to Program Optimization. MIT Press, Cambridge, MA, USA (2000)
15. CONVEX Computer Corp. O. Box 5937, Denver, CO 80217-9808: CONVEX Application Compiler User's Guide, 2nd ed. (1992)
16. Intel Corporation: Block-Matching In Motion Estimation Algorithms Using Streaming SIMD Extensions 3. Technical report, Intel Corporation, O. Box 5937, Denver, CO 80217-9808 (2003)
17. Pryanishnikov, I., Krall, A., Horspool, N.: Pointer Alignment Analysis for Processors with SIMD Instructions. In: 5th Workshop on Media and Streaming Processors. (2003)