Register Allocation for Irregular Architectures

Bernhard Scholz
Institute of Computer Languages
Vienna University of Technology, Austria
scholz@complang.tuwien.ac.at

Erik Eckstein
ATAIR Software GmbH
Vienna, Austria
Eckstein@atair.co.at

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compiler Code Generation Optimization*

General Terms

Algorithms Design Theory

Keywords

Register Allocation, Boolean Quadratic Problem

ABSTRACT

For irregular architectures global register allocation is still a challenging problem that has not been successfully solved so far. The graph-coloring analogy of traditional approaches does not match the needs of register allocation for such architectures which feature non-orthogonal instruction sets and small register files. This work proposes a fundamentally new approach to global register allocation for irregular architectures. Our approach formulates global allocation as a *partitioned boolean quadratic optimization problem* (PBQP) that allows generic modeling of processors peculiarities. Because PBQP is NP-complete we present a heuristic that exhibits a nearly linear run-time complexity.

We integrated our register allocator with the Infineon Carmel C Compiler which is based on the Open Compiler Environment from Atair Software. A DSP benchmark suite was used to compare the performance of our register allocator with a graph-coloring approach and with an optimal allocation. The experiments show that our new approach performs better than a traditional graph coloring approach for irregular architectures.

1. INTRODUCTION

Global register allocation is of vital importance for optimizing compilers. In the advent of embedded processors and in the predominance of legacy architectures register allocation algorithms have to cope with processor peculiarities such as non-orthogonal instruction sets and highly complex register models.

Traditionally, register allocation abstracts the problem of assigning CPU registers to symbolic registers[1] into the problem of coloring vertices in an *interference graph* [3, 2]. The vertices of the interference graph represent symbolic registers and edges of the graph imply interference constraints, i.e. two symbolic registers connected by an edge cannot be assigned the same CPU register. A coloring of this graph is equivalent to an assignment of CPU registers. If a symbolic register cannot be colored, the symbolic register must be *spilled* to memory, and additional load and store operations are inserted, which cause a performance loss. Graph coloring with a minimum number of colors is a well-known NP-complete problem and heuristics are applied to obtain an allocation in almost linear time [2, 3].

Although graph-coloring is a success story for RISC-like architectures with plenty of registers it cannot be simply adopted for irregular architectures. In the past several extension for graph-coloring were proposed [13, 9, 4] which fail to give a unified solution for irregular architectures. The most cunning problems are that (1) hardware restrictions are too conservatively modeled and graph-coloring results in poor allocations, (2) often irregularities increase the complexity of the interference graph which badly affects graph-coloring, (3) only a limited subset of restrictions can be modeled — but not all. Recently, register allocation based on integer linear programming (ILP) was introduced [7, 8]. This approach allows a precise modeling of irregularities and computes an optimal allocation. However, it is not practical since the underlying algorithms have exponential solve time.

To overcome the obstacles of register allocation for irregular architectures we present a new approach which exhibit nearly linear performance and can cope with irregularities. In contrast to traditional approaches, coalescing [12, 6] is no longer a separate pass – copy propagation is considered as integral part of the register assignment which is indispensable for a good register assignment.

Our approach works twofold. First, a mathematical model of the register allocation problem is built. The model comprises various aspects of registers allocation for regular and irregular architectures such as interferences, coalescing, and hardware peculiarities. Second, the model is mapped to

LCTES'02–SCOPES'02, June 19-21, 2002, Berlin, Germany.

---

[1] There might be a difference between a symbolic register and a live range, for sake of simplicity we assume that there is a one-to-one relationship between live-range and symbolic register. Therefore, we use the term *symbolic register* in this work.

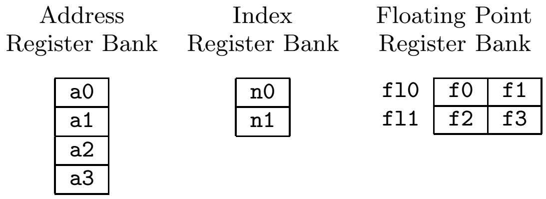


Figure 1: Hypothetical CPU Architecture

the Partitioned Boolean Quadratic Optimization Problem (PBQP) and solved by a polynomial solver. Although solving the model is inherently NP-complete our solver computes an allocation in nearly linear time by using heuristics.

We conducted experiments to demonstrate the effectiveness and efficiency of our new register allocator. As a testbed we used the Infineon Carmel C Compiler for the Carmel 20xx DSP Core which is a highly constrained architecture for register allocation. Kernels of typical DSP applications were employed to compare the performance of our new approach with a graph-coloring approach and with an optimal solution. The experiments show that our new approach achieves significantly better allocations in comparison with a traditional graph-coloring approach and the allocation results of our new approach is nearly as good as the optimal solution although there is potential for further improvements of the heuristic.

Our work goes beyond previous work in the following points: (1) unified approach for irregular architectures, (2) precise modeling of irregularities, (3) highly adaptable for irregular architectures, (4) coalescing as integral part of register assignment, (5) significant better results for irregular architectures.

The paper is organized as follows. The next section introduces some useful notations and definitions. In Section 3 we motivate our approach by an illustrative example. In Section 4 we discuss the modeling of register allocation for irregular architectures and we introduce an optimization problem for register allocation. In Section 5 we present a polynomial solver for the optimization problem. In Section 6 the performance and implementation details are given for the Infineon Carmel C-Compiler. Related work is surveyed in Section 7 and, finally, we draw our conclusions in Section 8.

2. BACKGROUND

Matrix Calculus. A *matrix* A is any rectangular array of elements. If a matrix A has m *rows* and n *columns*, we call A a $m \times n$ matrix. Let $A(i, j)$ be the ij-th element in the i-th row and j-th column. The i-th row is denoted by $A(i, :)$ and the j-th column by $A(:, j)$. A matrix in which all elements are zero is called *zero matrix*.

A *vector* $\vec{v}$ is a $1 \times n$ matrix with n elements. Let $\vec{v}(i)$ be the i-th element of the vector and let $|\vec{v}| = n$ be the length of a vector. Vector $\vec{1}$ in which all elements are one is called *one vector*. The minimum (written $\min(\vec{v})$) is the smallest element c of vector $\vec{v}$ such that $\forall 1 \leq i \leq |\vec{v}| : c \leq \vec{v}(i)$. The *minimum index* $i_{min}(\vec{v})$ of a vector is defined to be the smallest index of minima and the following must hold: $\vec{v}(i_{min}(\vec{v}))) = \min(\vec{v})$. Given any $m \times n$ matrix A, the *transpose* for A (written A^T) is the $n \times m$ matrix whose ij-th element $A^T(i, j) = A(j, i)$. Let A and B two $m \times n$ matrices. The matrix $C = A + B$ is defined to be the $m \times n$ matrix whose ij-th element is $C(i, j) = A(i, j) + B(i, j)$. The *matrix product* $C = AB$ of A and B is the $m \times n$ matrix whose ij-th element is determined by $C(i, j) = \sum_{k=1}^{l} A(i, k)B(k, j)$ where l is the number of columns in matrix A and the number of rows in matrix B.

```
sa0 = stack0
sa1 = stack1
sa2 = a2
sfl0 = 0
sn0 = 0
loop {
  sn1 = sn0
  sf1 = *(sa0 + sn1)
  sn2 = sn0
  sf2 = *(sa1 + sn2)
  sfl0 += sf1 * sf2
  sn0 += 1
}
sn3 = sn0
*(sa2 + sn3) = sfl0
```

Figure 2: Running example

A *quadratic form* is defined to be a vector-matrix-vector product $\vec{x}A\vec{y}^T$, which has the following properties:

$$\begin{aligned} \vec{x}A\vec{y}^T &= \vec{y}A^T\vec{x}^T \\ \vec{x}A\vec{y}^T + \vec{x}B\vec{y}^T &= \vec{x}\,(A + B)\,\vec{y}^T \end{aligned}$$

Graphs. Let $G\langle V, E\rangle$ be a *directed graph*, where V is a set and E a relation on V. The elements of V are called vertices and the ordered pairs in E are called edges. A graph is *undirected* if E is symmetric, i.e. $(u, v) \in E \Leftrightarrow (v, u) \in E$. Removing edge (u, v) from a symmetric graph is denoted by $E - (u, v)$, which is equivalent to $E - \{(u, v), (v, u)\}$. Similar accounts for adding an edge: $E + (u, v) = E \cup \{(u, v), (v, u)\}$.

Let $adj(u) = \{v | (u, v) \in E\}$ be a set of *adjacent vertices* of u and $deg(u) = |\, adj(u)\, |$ the *degree* of vertex u. A vertex u is said to be *disconnected* if $deg(u) = 0$.

3. EXAMPLE

In this section we motivate our approach by presenting a hypothetical irregular architecture and a running example that computes an inner dot-product of two vectors. The irregular architecture has similarities with an embedded system like CPU architecture consisting of various register classes as depicted in Figure 1. Our hypothetical CPU architecture attributes four register classes where some of the registers are shared. Namely, it has four address registers in the address register bank, two index registers in the index register bank, and two 64bit floating point register. Moreover, each 64bit floating point register can either be accessed as one 64bit register or as two 32bit floating point registers.

The addressing unit of our hypothetical CPU architecture supports a register indexed addressing mode that computes the memory address by adding an address register and an index register. However, index register `n0` can only be paired with address registers `a0` and `a1` – similar accounts for index register `n1`, which can only be paired with address registers `a2` and `a3`. The following scheme shows the allowed pairs of address and index registers:

Figure 2 lists our running example. In the example we have symbolic registers for each registers class, i.e. symbolic address registers are denoted as `sa`i, symbolic index registers as `sn`i, floating point registers as `sf`i, and long floating point registers as `sfl`i. Our example consists of a prologue (line 1–5), a loop (line 6–13), and an epilogue (line 14–15). As shown in line 1–3 we have a mixed argument passing for symbolic address register `sa0`, `sa1`, and `sa2`. Symbolic address register `sa2` is initialized by an argument passed in the CPU register `a2`, the others are initialized from the calling stack. In the loop two floating point values are read from memory locations (line 8 and line 10). For the memory accesses an indexed addressing mode `*(sa`i` + sn`j`)` is used. Line 11 contains the accumulate instruction of the dot product and line 12 increments the index register for memory locations. In the epilogue the result is stored in memory by using the indexed addressing mode.

The task of register allocation is to assign CPU registers to symbolic registers and to spill symbolic registers to memory if not enough CPU registers are available. The allocator has to pursue two goals: (1) the number of expensive spills should be as small as possible and (2) it should perform copy propagation which is achieved by giving source and target of a copy instruction the same register whereby the copy instruction can be removed afterwards [12].

The move instructions in line 7, 9 and 14 are inserted by the code generator before allocating registers. This is done since the indexed addressing mode constraints the selection of registers and, in general, a copy instruction is cheaper than a spill.

The register allocator should aim to eliminate the copy instructions by assigning the source and destination to same register. However eliminating moves in line 7, 9, and 14 would imply a pairing of a single index register to three different address registers (indexed addressing modes in line 8, 10 and 15). In our architecture we can only pair two address register to one index register. Therefore at least one move instruction cannot be eliminated — preferable the move instruction that is outside of the loop (line 14).

Our hypothetical architecture exhibits another constraint for register allocation called *register sharing*. Two short floats share the same memory of one long float. These registers are no longer independent of each other and the register allocator has to take care of it. E.g., our running example has three symbolic floating point registers (one long and two short floats) and the allocator should pack registers `sf1` and `sf2` into one long floating point register — otherwise symbolic float registers must be spilled to memory.

4. REGISTER ALLOCATION

For register allocation it is of paramount importance to have an accurate model of cost to obtain a good decision which symbolic registers are to be spilled and which symbolic registers are stored in CPU registers. In contrast to RISC-like architectures, irregular architectures exhibit a non-uniform cost model. Non-uniform costs stem from various architectural peculiarities and special care must be taken to design the model of cost [8].

In our approach we employ cost functions for register assignment decisions. The cost functions have to fulfill two tasks: (1) cost functions mathematical express the model of cost of the architecture, and (2) cost functions are used to describe interference constraints, coalescing benefits, and constraints which stem from the CPU architecture (e.g. register pairing, etc.).

4.1 Cost Functions

Basically, we have two classes of cost functions. One class of cost functions model the costs and constraints involved for one symbolic register, and the second class of cost functions model the costs and constraints of two dependent symbolic registers. Both classes are sufficient for modeling register allocation on common irregular architectures.

Definition 1. *A cost function of symbolic register* `s` *is a function* $f_{\mathtt{s}} : A \rightarrow \mathbb{R} \cup \{\infty\}$ *where* $A = \{sp, R_1, R_2, \ldots, R_m\}$ *is an assignment set and* $\mathbb{R}$ *is the set of real numbers.*

Cost function $f_{\mathtt{s}}$ expresses the costs either for assigning symbolic register `s` to one of the CPU registers $R_1, \ldots, R_m$ or for spilling symbolic register `s` (denoted by $sp \in A$). If assignment $a \in A$ is not valid for `s`, cost function $f_{\mathtt{s}}$ maps assignment a to ∞. There might be more than one cost function for a symbolic register `s` and we aggregate them in a set of cost functions denoted by $F_{\mathtt{s}}$.

Definition 2. *A cost function of a pair of symbolic registers* $\mathtt{s}_i$ *and* $\mathtt{s}_j$ *is a function* $f_{\mathtt{s}_i\mathtt{s}_j} : A \times A \rightarrow \mathbb{R} \cup \{\infty\}$.

Cost function $f_{\mathtt{s}_i\mathtt{s}_j}$ expresses the costs for two dependent symbolic registers. If a combination of two register assignments $a_1, a_2 \in A$ for $\mathtt{s}_i$ and $\mathtt{s}_j$ is not valid, the cost function maps a_1 and a_2 to ∞. For every pair of symbolic register we have a set of cost functions denoted by $F_{\mathtt{s}_i\mathtt{s}_j}$.

In the following we want to consider some typical kinds of cost functions by looking at our running example. For modeling spilling costs we introduce function $s^{(c)}_{\mathtt{s}}(a)$:

$$s^{(c)}_{\mathtt{s}}(a) = \begin{cases} c, & \text{if } a = sp \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

If `s` is spilled($a = sp$), function $s^{(c)}_{\mathtt{s}}(a)$ maps assignment a to c. The value c denotes the costs of spilling of symbolic register `s`. If a register is assigned to `s` ($a \in \{R_1, \ldots, R_m\}$), function $s^{(c)}_{\mathtt{s}}(a)$ has zero costs.

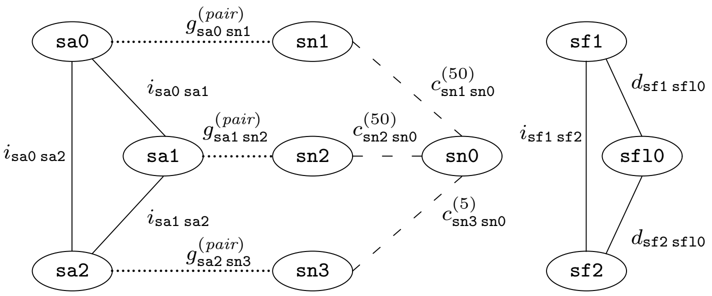
For example we have in our architecture 12 CPU registers and following assignment set:

$$A = \{sp, \mathtt{sa0}, \ldots, \mathtt{sa3}, \mathtt{sn0}, \mathtt{sn1}, \mathtt{sf0}, \ldots, \mathtt{sf3}, \mathtt{sfl0}, \mathtt{sfl1}\}$$

Say, the loop is performed 10 times and `sa0` is accessed 11 times. Therefore, we insert $s^{(110)}_{\mathtt{sa0}}$ into the function set $F_{\mathtt{sa0}}$ by assuming that a read accesses costs 10 units if `sa0` is spilled.

Another important constraint is the interference constraint of two symbolic registers. The interference constraint is represented by a cost function for two symbolic registers as follows:

$$i_{\mathtt{s}_1\mathtt{s}_2}(a_1, a_2) = \begin{cases} 0, & \text{if } a_1 \neq a_2 \vee a_1 = sp \\ \infty, & \text{otherwise} \end{cases} \tag{2}$$



s	$F_{\mathtt{s}}$
sa0	$\{s^{(110)}_{\mathtt{sa0}}, c_{\mathtt{sa0}}\}$
sa1	$\{s^{(110)}_{\mathtt{sa1}}, c_{\mathtt{sa1}}\}$
sa2	$\{s^{(110)}_{\mathtt{sa2}}, c_{\mathtt{sa2}}, r^{(5,\mathtt{a2})}_{\mathtt{sa2}}\}$
sn0	$\{s^{(110)}_{\mathtt{sn0}}, c_{\mathtt{sn0}}\}$
sn1	$\{s^{(100)}_{\mathtt{sn1}}, c_{\mathtt{sn1}}\}$
sn2	$\{s^{(100)}_{\mathtt{sn2}}, c_{\mathtt{sn2}}\}$
sn3	$\{s^{(100)}_{\mathtt{sn3}}, c_{\mathtt{sn3}}\}$

s	$F_{\mathtt{s}}$
sfl0	$\{s^{(220)}_{\mathtt{sfl0}}, c_{\mathtt{sfl0}}\}$
sf1	$\{s^{(200)}_{\mathtt{sf1}}, c_{\mathtt{sf1}}\}$
sf2	$\{s^{(200)}_{\mathtt{sf2}}, c_{\mathtt{sf2}}\}$

Figure 3: Cost functions of running example

The cost function implies that symbolic registers $\mathtt{s}_1$ and $\mathtt{s}_2$ must be assigned different CPU registers. In our example all symbolic address registers interfere with each other. Therefore we have cost functions $i_{\mathtt{sa0\,sa1}}$, $i_{\mathtt{sa0\,sa2}}$, and $i_{\mathtt{sa1\,sa2}}$ which are inserted to the corresponding function sets $F_{\mathtt{sa0\,sa1}}$, $F_{\mathtt{sa0\,sa2}}$, and $F_{\mathtt{sa1\,sa2}}$.

Irregular CPU architectures often have various register classes. The register allocator must take care which CPU register is a valid assignment for a symbolic register. More formally, a class *class*(s) is a sub-set of valid CPU registers for symbolic register s and the cost function is given as follows:

$$c_{\mathtt{s}}(a) = \begin{cases} 0, & \text{if } a \in \mathit{class}(\mathtt{s}) \cup \{\mathit{sp}\}, \\ \infty, & \text{otherwise} \end{cases} \tag{3}$$

The symbolic registers of our example also have dedicated register classes. For sake of simplicity there is a one to one relation ship between symbolic register classes and CPU register classes in the example. E.g., symbolic register `sa0` can only be assigned one of the registers `a0`. . . `a3`. The class constraint of `sa0` is expressed as $c_{\mathtt{sa0}}$ and it needs to be added to $F_{\mathtt{sa0}}$. However, it is straight forward to model different register classes for a symbolic registers since every register of the register set can either be selected or omitted by set *class*(s) of symbolic register s.

Copy propagation is achieved by eliminating an copy instruction if source and target are assigned the same CPU register. The costs for eliminating a copy instructions are expressed as cost function of source and destination register,

$$p^{(b)}_{\mathtt{s}_1\,\mathtt{s}_2}(a_1, a_2) = \begin{cases} -b, & \text{if } a_1 = a_2 \wedge a_1 \neq \mathit{sp} \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

where b is the benefit of eliminating the copy instruction. In our example lines 7, 9 and 14 contain copy instructions. We assume that we save 5 cost units by eliminating a copy instruction, and that the loop is executed 10 times. Then, for copy propagation the following cost functions are imposed: $p^{(50)}_{\mathtt{sn1\,sn0}}$, $p^{(50)}_{\mathtt{sn1\,sn0}}$, and $p^{(5)}_{\mathtt{sn1\,sn0}}$. For the first two copy instructions we have 50 cost units since the loop is executed 10 times. For eliminating the copy instruction in the epilogue 5 cost units are saved. Again, the cost functions need to be added to the corresponding function sets $F_{\mathtt{sn1\,sn0}}$, $F_{\mathtt{sn1\,sn0}}$, and $F_{\mathtt{sn1\,sn0}}$.

Copy propagation is also relevant for parameter passing when parameters are passed via registers. In this case one operand of the copy instruction is a CPU register rather than a symbolic register. The copy instruction can be eliminated if the symbolic register is assigned to the same CPU register. The following cost function models the benefit b if the copy instruction can be eliminated:

$$r^{(b,R_c)}_{\mathtt{s}}(a) = \begin{cases} -b, & \text{if } a = R_c, \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

where b is the benefit of eliminating the copy instruction and R_c is the CPU register in which the parameter is stored. In line 3 of our example symbolic register `sa2` is assigned to a parameter which is held in `a2`. If symbolic register `sa2` is stored in CPU register `a2`, the copy instruction can be eliminated. Let us assume that we need 5 cost units for a copy instruction. Then, the cost function for `sa2` is given by $r^{(5,\mathtt{a2})}_{\mathtt{sa2}}$.

Shared registers[2] are very common in irregular architectures. Registers which share the same memory can not be used at the same time. Shared registers impose an interference constraint between symbolic registers although they belong to different register classes. For every CPU register $a \in \{R_1, \ldots, R_m\}$ we have a set of shared registers *shared*(a). In order to keep the cost function simple we extend the set of shared registers for spilling: (1) $\forall a \in \{R_1, \ldots, R_m\} : \{a, \mathit{sp}\} \subseteq \mathit{shared}(a)$ and (2) $\mathit{shared}(\mathit{sp}) = \{R_1, \ldots, R_m\}$. Then, the cost function is given as follows:

$$d_{\mathtt{s}_1\,\mathtt{s}_2}(a_1, a_2) = \begin{cases} \infty, & \text{if } a_1 \in \mathit{shared}(a_2) \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

In our architecture long floating point registers (`fl0`,`fl1`) share the same memory with short floating point registers (`f0`. . . `f3`). Therefore, we have an interference between `sfl0` and `sf1`, and an interference between `sfl0` and `sf2`, which are denoted by $d_{\mathtt{sfl0\,fl1}}$ and $d_{\mathtt{sfl0\,fl1}}$.

Irregular architecture often impose constraints for instruction operands for keeping the instruction word small. E.g., in many DSP architectures only a sub-set of index registers are allowed in conjunction with an address register. We model generic dependencies of two symbolic register operands by the following cost function

$$g^{(\alpha)}_{\mathtt{s}_1\mathtt{s}_2}(a_1, a_2) = \begin{cases} 0, & \text{if } a_1 \in \alpha(a_2) \\ \infty, & \text{otherwise} \end{cases} \tag{7}$$

---

[2] In [8] the term *overlapped registers* is used for shared registers.

where $\alpha(a_2)$ is a set of allowed register combinations for assignment a_1. E.g., the indexed addressing accesses of our running example in line 8, 10 and 15 is such a constraint. In our architecture we can only pair `a0` and `a1` with `n0`, and `a2` and `a3` with `n1`. For the pairing constraint we introduce a set *pair*(a) defined as follows:

a	*pair*(a)
sp	$\{sp, \mathtt{a0}, \dots, \mathtt{a3}\}$
`n0`	$\{sp, \mathtt{a0}, \mathtt{a1}\}$
`n1`	$\{sp, \mathtt{a2}, \mathtt{a3}\}$

The set *pair*(a) contains allowed address registers for a symbolic index register. Based on set *pair*(a) we can precisely express the constraints of the indexed addressing mode that is used in line 8, 10, and 15. For our running example we have the following cost functions: $g^{(pair)}_{\mathtt{sa0}\,\mathtt{sn1}}$, $g^{(pair)}_{\mathtt{sa1}\,\mathtt{sn2}}$, and $g^{(pair)}_{\mathtt{sa2}\,\mathtt{sn3}}$.

Now we are able to put together the cost functions for our running example. Figure 3 depicts a graph on the left-hand side that shows the cost functions between two dependent symbolic registers. In the graph interference constraints are drawn by straight lines, copy propagation constraints by dashed lines, and generic hardware constraints by dotted lines. The graph shows the interference constraints between all symbolic floating point registers although the type of interference differs: There is an interference of shared registers between short floats `sf1`, `sf2` and long float `sfl0`, and a classical interference between `sf1` and `sf2`. The interferences of address registers are depicted as well and due to the indexed addressing modes, pairing constraints are drawn between address registers (`sa0`. . . `sa2`) and indexed registers (`sn1`. . . `sn3`). Copy propagation constraints are drawn between symbolic index register `sn0` and registers `sn1`. . . `sn3`.

On the right-hand side of Figure 3 a table lists function sets for all symbolic registers of our running example. All sets contain cost functions for spilling and register classes. For symbolic register `sa2` the function set also contains the copy propagation for the register argument.

We can see that even for simple codes the constraints imposed by the architecture, can become quite complex. In contrast to our approach, traditional graph-coloring techniques fail to precisely express costs and architectural constraints for register assignments.

4.2 Optimization Problem

In the following we map the set of cost functions to an NP-complete optimization problem called *boolean quadratic optimization problem* [14, 5].

For a symbolic register the allocator has to decide whether it stores the symbolic register in one of the CPU registers $R_1, \dots, R_m$ or it spills the symbolic register. We express this principle decision by a boolean equation which constitutes of a set of boolean variables. One of the boolean variables represents the decision that the symbolic register is spilled. The remaining boolean variables represent the decisions with which CPU register the symbolic register is associated with. Following mathematical formula forms this principle decision:

$$x_{sp} + x_{R_1} + x_{R_2} + \ldots + x_{R_m} = 1, \tag{8}$$

where $x_{sp} \in \{0, 1\}$ is the boolean variable for spilling and $x_{R_i} \in \{0, 1\}$ is the boolean variable for the i-th CPU register.

The boolean equation can be rewritten as a matrix product of a boolean vector $\vec{x}_{\mathtt{s}}\,\vec{1}^T = 1$, where the elements of $\vec{x}_{\mathtt{s}}$ are the boolean variables and $\vec{1}$ is the one vector. An assignment $a \in A$ for symbolic register `s` corresponds to an index of boolean vector $\vec{x}_{\mathtt{s}}$ whose corresponding boolean variable is set to 1.

The set of cost functions of one symbolic register `s` can be aggregated to cost vector $\vec{c}_{\mathtt{s}}$,

$$\forall a \in A : \vec{c}_{\mathtt{s}}(\phi_a) = \sum_{f_{\mathtt{s}} \in F_{\mathtt{s}}} f_{\mathtt{s}}(a) \tag{9}$$

where ϕ_a is the index of assignment a.

For example the cost vector of symbolic register `sa2` is composed by its cost functions:

$$\vec{c}_{\mathtt{sa2}}(\phi_a) = \sum_{f_{\mathtt{sa2}} \in F_{\mathtt{sa2}}} f_{\mathtt{sa2}}(a) = s^{(110)}_{\mathtt{sa2}}(a) + c_{\mathtt{sa2}}(a) + r^{(5,\mathtt{a2})}_{\mathtt{sa2}}(a)$$

Let us assume that assignment a is mapped to indices ϕ_a by the following order: *sp*, `sa0`, . . . , `sa3`, `sn0`, . . . , `sn3`, `sf0`, . . . , `sf3`, `sfl0`, `sfl1`. We obtain the following cost vector for symbolic register `sa2`:

$$\vec{c}_{\mathtt{sa2}} = (110, 0, 0, -5, 0, \infty, \dots, \infty)$$

The first element of vector $\vec{c}_{\mathtt{sa2}}$ gives spill costs of symbolic register `sa2` and is the contribution of function $s^{(110)}_{\mathtt{sa2}}(a)$. The second to the 5th element indicate an allowed assignment whereas the remaining elements are disabled by class constraint $c_{\mathtt{sa2}}(a)$ since symbolic address register `sa2` can only be stored in a CPU address register (`a0`. . . `a3`). The forth element of the cost vector is the cheapest assignment due to cost function $r^{(5,\mathtt{a2})}_{\mathtt{sa2}}(a)$ for eliminating copy instruction in line 3 of our example.

To model costs of two dependent symbolic registers $\mathtt{s}_1$ and $\mathtt{s}_2$ we use quadratic forms $x_{\vec{\mathtt{s}}_1} C_{\mathtt{s}_1\,\mathtt{s}_2} x_{\vec{\mathtt{s}}_2}{}^T$, where $C_{\mathtt{s}_1\,\mathtt{s}_2}$ is a cost matrix. The cost matrix is computed by function sets as follows:

$$\forall a_1, a_2 \in A : C_{\mathtt{s}_1\,\mathtt{s}_2}(\phi_{a_1}, \phi_{a_2}) = \sum_{f_{\mathtt{s}_1\,\mathtt{s}_2} \in F_{\mathtt{s}_1\,\mathtt{s}_2}} f_{\mathtt{s}_1\,\mathtt{s}_2}(a_1, a_2) \tag{10}$$

For example an interference constraint of `sa0` and `sa1` impose the following cost matrix:

$$C_{\mathtt{sa0}\,\mathtt{sa1}} = \begin{pmatrix} 0 & 0 & \dots & 0 \\ 0 & \infty & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & \infty \end{pmatrix}$$

The cost matrix weights pairs of boolean variables which belong to the same register with infinite costs. All other pairs of boolean variables, i.e. for different registers or spilling, have zero costs.

For sake of simplicity we have an order of symbolic registers and we refer to symbolic registers with their unique ordinal number. E.g., `sa0` is the first symbolic registers and the boolean vector of $\vec{x}_{\mathtt{sa0}}$ is rewritten as $\vec{x}_1$. We do the same for cost vectors and matrices to simplify the formula. Now, we are able to express register allocation as a boolean quadratic optimization problem.

Definition 3. *A partitioned boolean quadratic problem (PBQP) is defined as follows,*

$$\textit{min } f = \left[\sum_{1\leq i<j\leq n} \vec{x}_i \cdot C_{ij} \cdot \vec{x}_j^T\right] + \left[\sum_{1\leq i\leq n} \vec{c}_i \cdot \vec{x}_i^T\right] \tag{11}$$

$$\textit{subject to: } \forall i \in 1 \ldots n : \vec{x}_i \cdot \vec{1}^T = 1 \tag{12}$$

where n is the number of symbolic registers[3].

Due to the symmetric properties of quadratic forms the objective function is a triangular sum. Note that there exists at least one solution which has less than infinity costs as long as the spill costs are finite. This is due to the fact that all symbolic registers can be spilled although this solution is not a good one.

5. SOLVER

For solving PBQP we employ dynamic programming. By its nature the problem is NP-complete and we do not get an optimal solution in polynomial time. However, heuristics are applied in order to obtain a satisfying solution.

Our dynamic programming approach works in three phases. In the first phase we reduce the PBQP into subproblems. In each reduction dynamic programming eliminates a decision vector until the objective function becomes trivial:

$$f = \sum_{1\leq i\leq n} \vec{c}_i \cdot \vec{x}_i^T \tag{13}$$

In the second phase we determine the solution for decision vectors in f. The solutions of the decision vectors $\vec{x}_i$ can be determined by finding the smallest element of $\vec{c}_i$ since there is no dependence between decision vectors. Therefore, solution of $\vec{x}_i$ is $s_i = i_{min}(\vec{c})$ where s_i is the index of the boolean variable in $\vec{x}$ whose value is set to one and relates to a register assignment.

In the third phase we compute solutions for decision vectors which were eliminated in the first phase by propagating the solution through the eliminated decision vectors.

The puzzling questions is to find reduction rules, which can be performed in polynomial time such that the optimality of PBQP is not destroyed. In general this is not possible since the underlying problem is NP-complete. For sake of computability we have to sacrifice optimality for some decision vectors. We call these decision vectors *irreducible* and employ heuristics to reduce them. Other decision vectors are *reducible* and they do not destroy optimality.

For the algorithm we construct an undirected *PBQP graph* $G\langle V, E, w\rangle$. The PBQP graph is an equivalent representation of the PBQP and allows a linear selection of reduction rules for reducible decision vectors. In the PBQP graph decision vector $\vec{x}_i$ is represented by a vertex $v_i \in V$ $(1 \leq i \leq n)$. Vertices $v_i \in V$ and $v_j \in V$ $(1 \leq i < j \leq n)$ are connected by an edge $(v_i, v_j) \in E$ if cost matrix C_{ij} is not zero. Cost function w maps vertex $v_i \in V$ to cost vectors $\vec{c}_i$ and edge $(v_i, v_j) \in E$ to cost matrix C_{ij}. Cost function w distinguishes the direction of an edge in the undirected graph: cost matrix C_{ij} of edge $(v_i, v_j) \in E$ is the transposed matrix of edge (v_j, v_i): ${C_{ij}}^T = C_{ji}$ and vice versa. Note that this direction aware cost function reflects the properties of quadratic forms as introduced in Section 2. In the following we denote C_{uv} as cost matrix of edge (u, v) which is equivalent with w(u,v). Due to the structure of PBQPs, G has no reflexive edge $(v_i, v_i) \notin E$, since we have no cost matrices C_{ii} in objective function f. For our running example the graph has been already given in Figure 3 on the left-hand side.

Our PBQP solver employs three *reduction rules* which are RI, RII, and RN. We denote the original graph as Graph G^0. Each application of a reduction rule transforms some reduced PBQP graph G^i to G^{i+1} until all vertices of the resulting graph are disconnected, i.e. the objective function becomes trivial. An application of reduction rule RI or RII transforms graph G^i into a subproblem without destroying the optimality of the objective function and they can only be applied for reducible vertices. Rule RN is applied to irreducible vertices and it can destroy the optimality of the objective function.

Figure 4 lists the Procedure *ReduceGraph* that is responsible for the first phase of dynamic programming. To select reducible vertices in constant time, *buckets* are used. For each vertex degree there exists a bucket and a vertex belongs to bucket d if it has degree d. Note that the first three lines of Procedure *ReduceGraph* build up the buckets. After constructing the buckets, vertices are reduced until all vertices are in bucket 0 and the objective function becomes trivial.

Definition 4. *Let $G^i\langle V, E, w\rangle$ be the i-th reduced PBQP graph. Let x be a vertex of degree one and y the adjacent vertex of x. Then*
$RI(G^i\langle V, E, w^i\rangle, x) \Rightarrow G^{i+1}\langle V - \{x\}, E - (x, y), w^{i+1}\rangle$

The RI-rule is applied to a vertex x of degree one. The weighting function w^{i+1} is identical to weighting function w^i except for vertex y. The cost vector $\vec{c}_y$ is incremented by $\vec{\Delta}$ as shown in procedure *ReduceI*. Vector $\vec{\Delta}$ considers minimal costs of vertex x dependent on the decision of y. After incrementing $\vec{c}_y$, Procedure PushVertex(x) pushes vertex x onto a stack, and removes it from the PBQP graph. Then, it reorders adjacent vertices in the buckets according to their degrees.

Definition 5. *Let $G^i\langle V, E, w\rangle$ be the i-th reduced PBQP graph. Let x be a vertex of degree two and y and z adjacent vertices of x. Then*
$RII(G^i\langle V, E, w^i\rangle, x) \Rightarrow$
$G^{i+1}\langle V - \{x\}, E + (y, z) - (x, y) - (x, z)\}, w^{i+1}\rangle$

The RII-rule is applied to vertex x if x has degree of two. Procedure *ReduceII* in Figure 4 performs the RII-rule for vertex x. The weight of the edege (y, z) is determined by the Δ matrix as computed in Procedure ReduceII. If edge (y, z) already exists in the graph, we add Δ to cost matrix C_{yz}. Otherwise, a new edge (y, z) is inserted and the edge (y, z) is assigned the cost matrix Δ as new weight. Then, Procedure PushVertex(x) pushes vertex x onto a stack, and removes the vertex from the graph.

Definition 6. *Let $G^i\langle V, E, w\rangle$ be the i-th reduced PBQP graph. Let x be a vertex. Then*
$RN(G^i\langle V, E, w\rangle, x) \Rightarrow$
$G^{i+1}\langle V - \{x\}, E - \sum_{y\in \mathrm{adj}(x)}(x, y), w'\rangle$

---

[3] In the PBQP problem we redefine the semantics of ∞. Let $\infty \cdot 0 = 0$.

```
1:  procedure ReduceI($x$)
2:  begin
3:      $\{y\} := adj(x)$;
4:      for i:=1 to $|\vec{c}_y|$ do
5:          $\vec{\Delta}(i) := \min(C_{yx}(i,:) + \vec{c}_x)$;
6:      endfor
7:      $\vec{c}_y := \vec{c}_y + \vec{\Delta}$;
8:      PushVertex(x);
9:  end
10: procedure ReduceII($x$)
11: begin
12:     $\{y,z\} := adj(x)$;
13:     for $i := 1$ to $|\vec{c}_y|$ do
14:         for $j := 1$ to $|\vec{c}_z|$ do
15:             $\Delta(i,j) := \min(C_{yx}(i,:) + C_{zx}(j,:) + \vec{c}_x)$;
16:         endfor
17:     endfor
18:     if $(y,z) \in E$ then
19:         $C_{yz} := C_{yz} + \Delta$;
20:     else
21:         add edge $(y,z)$
22:         $C_{yz} := \Delta$;
23:     endif
24:     PushVertex(x)
25: end
26: procedure ReduceN($x$)
27: begin
28:     for $i := 1$ to $|\vec{c}_x|$ do
29:         $\vec{c}(i) := 0$;
30:         forall vertices $y \in adj(x)$ do
31:             $\vec{c}(i) := \vec{c}(i) + \min(C_{xy}(i,:) + \vec{c}_y)$;
32:         endfor
33:     endfor
34:     for $i := 1$ to $|\vec{c}_x|$ do
35:         if $s_x$ equal $i$ overconstraints adj. vertices then
36:             $\vec{c}(i) := \infty$;
37:         endif
38:     endfor
39:     $s_x = i_{min}(\vec{c})$;
40:     forall vertices $y \in adj(x)$ do
41:         replace $\infty$ by spill costs divided by
42:          $\deg(x)$ in $C_{xy}(s_x,:)$ and store result in $\vec{c}$
43:         $\vec{c}_y := \vec{c}_y + \vec{c}$;
44:     endfor
45:     PushVertex(x);
46: end
47: procedure ReduceGraph
48: begin
49:     forall vertices $x \in V$ do
50:         insert $x$ into bucket $\deg(x)$
51:     endfor
52:     while vertices left (buckets $\geq 1$) do
53:         if a vertices $x$ exists in bucket 1 then
54:             ReduceI($x$);
55:         elsif a vertices $x$ exists in bucket 2 then
56:             ReduceII($x$);
57:         else
58:             select vertex $x$
59:             ReduceN(x);
60:         endif
61:     endwhile
62: end
```

Figure 4: Reductions

```
procedure PropagateSolution
begin
    forall vertices $x$ in bucket 0 do
        $s_x := i_{min}(\vec{c}_x)$
    endfor
    while reducible stack not empty do
        pop vertex $x$ from reducible stack
        $\vec{c} := \vec{c}_y$;
        forall vertices $y \in adj(x)$ do
            $\vec{c} := \vec{c} + C_{yx}(:, s_x)$;
        endfor
        $s_x = i_{min}(\vec{c})$;
    endwhile
end
```

Figure 5: Propagate solution

The RN-rule is applied to irreducibles vertices. The rule is based on a heuristic that works as follows: It finds a local minimum for vertex x and distributes costs involved for the assignment to adjacent vertices. If some assignments for the adjacent vertices are forbidden due to the selected solution for x the corresponding costs would be infinite. Note that the selection of the local minimum of x is only an intended solution and it would give poor solutions for the whole PBQP. Therefore, we replace the infinite costs caused by the selection of x by the spilling costs of x divided by the degree of x. The local minium is not always a good selection for vertex x, since it can lead to situations that for adjacent vertex it is more expensive to assign a CPU register than spilling. To avoid this overconstraint adjacent vertices we disable this solution as shown in line 35 of Procedure *ReduceN*.

The algorithm has to chose irreducible vertices from a bucket of degree higher than two. The selection heavily influences the quality of the solution. In our implementation we used Briggs's scheme [2] for choosing irreducible vertices. The node with minimal cost-degree ratio is selected.

When the objective function is trivial we can determine solutions for vertices in bucket 0. In the sequel we propagate the solution through the eliminated vertices and reconstruct the graph. The second and the third phase of the dynamic program are given in Procedure *PropagateSolution* of Figure 5.

Our algorithm parititions vertices in four subsets of V: (1) a partition V_1 of vertices that are solved trivially in the objective function, (2) a partition V_2 of vertices reduced by rule RI, (3) a partition V_3 of vertices reduced by rule RII, and (4) a partition V_4 of vertices reduced by rule RN. All four partitions exhibit a different time complexity. We denote n_{V_i} the number of elements in the i-th partition whereas the sum is equal to n, i.e. the number of vertices. Then, the time complexity is given by

$$O(n_{V_1} \cdot m + n_{V_2} \cdot m^2 + n_{V_3} \cdot m^3 + n \cdot n_{V_4} \cdot m^2) \qquad (14)$$

where m is the number of registers. For a given architecture the number of CPU register is constant and we can see that all reductions have linear impact — except reduction RN which has a quadratic time complexity.

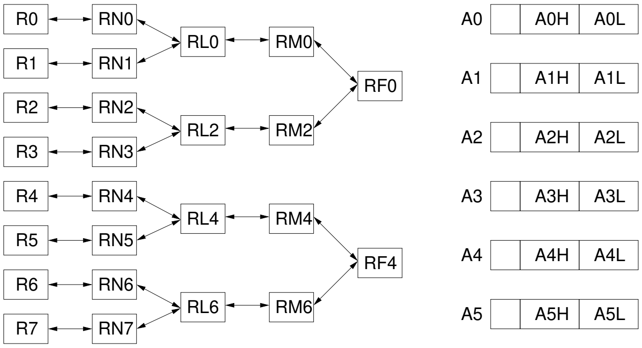


Figure 6: Carmel register model

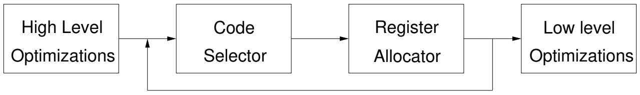


Figure 7: Compiler Framework

Name	Description
biq	bi-quadratic filter
fft	fast fourier transformation
hdvd	hard decision viterbi decoder
mmult	matrix multiplication
vit	viterbi filter

Table 1: DSP Benchmark

6. EXPERIMENTAL RESULTS

In order to demonstrate the efficiency and effectiveness of our new approach we have implemented it in the Infineon Carmel C Compiler and compared it with a classical approach [2, 13] for a DSP benchmark suite. The Infineon Carmel C Compiler produces assembly code for the Carmel 20xx DSP Core [11] which is a family of 16-bit, fixed-point DSPs that target infrastructure and hand-held devices for wired and wireless communications. The Carmel architecture features six arithmetic units and two data addressing units. The core can execute up to six instructions in parallel with a configurable long instruction word (CLIW). It has six accumulators, which are 40 bits wide and can also be used as 32 bit registers or as two 16 bit registers (see on the right-hand side of Figure 6). Additionally there are eight address registers, which can also be used as general purpose registers. The address registers (r0-r7) are 16 bit wide and have a whole set of supporting registers: index registers (rn0-rn7), modulo bound registers (rl0-rl6, rm0-rm6) and offset registers (rf0, rf4). The supporting registers are paired in a fixed schema to the address registers as shown on the left-hand side in Figure 6.

As typical for high-performance DSPs the Carmel register model introduces a large set of constraints for register allocation. E.g. accumulator registers (16bit, 32bit, or 40bit) impose constraints upon shared registers whereas address registers with their supporting registers demand pairing constraints as introduced in Section 4. Due to the complex register model it is a hard job to write a well-performing register allocator with traditional techniques.

The Infineon Carmel C Compiler features DSP C extension and is implemented with the Open Compiler Environment from Atair Software [10]. It contains a rich set of optimizations – some of them explicitly dedicated to utilize the Carmel architecture. Although the compiler can produce almost hand optimized code quality, the original register allocator with coalescing (based on traditional techniques [13]) has problems in handling pairing constraints and performing well for eliminating copy instructions. Therefore we were highly motivated to implement our new approach in this compiler environment. The integration of our register allocator was much simpler than the original register allocator since the architecture constraints were modeled with ease.

Figure 7 shows the involved passes for register allocation in the Carmel Compiler. After performing all high level optimizations the code selector produces low level code with symbolic registers. The code selector already decides in which register class a variable should be allocated (accumulators, half accumulators, address registers, index/offset registers or modulo registers). Then, the register allocator is invoked. If the register allocator decides to spill some registers, the code selector is reinvoked because the Carmel DSP core is not a load/store architecture and memory addresses can be used as operands. After allocating registers there are other passes which perform low level optimizations and scheduling.

We used C codes as a benchmark (see Table 1) which are very common for DSP applications, e.g. fast fourier transformation (fft), matrix multiplication (mmult), viterbi filter (vit), etc. In Table 2 we provide quantitative information for address and accumulator registers. The columns of the table contains the number of registers (Registers), the number of interference constraints (Interferences), and the number of coalescing edges (Copy Prop.). For address registers we give additional information about the number of pairing constraints (Pairing). Statistics in Table 3 show the allocation results of our new approach in comparison with a graph coloring approach [13, 2] and with an optimal allocation. The optimal allocation was obtained by adopting our algorithm. All possible decisions of RN-nodes were recursively enumerated which caused exponential computation time[4]. To compare our approach with an optimal allocation and with a traditional approach we provide for each performance number three columns in the Table 3. The first column (oPBQP) gives the optimal allocation, the second column (hPBQP) gives an heuristic solution as presented in Section 5, and third column (GrCo) gives a solution that was obtained by a traditional approach with graph coloring and coalescing. The runtime of the graph coloring algorithm and the heuristic of PBQP was negligible (i.e. both algorithms work linear in practice). In the table we present qualitative information about the number of spills (Spills). The number of spills hardly differ for all approaches since there are sufficient CPU registers. However, the allocation results of our approach supersedes traditional graph coloring algorithms due to the handicap of modeling hardware peculiarities and coalescing accurately. The spills cause loads and stores which results in a higher execution time and the register allocator has to decide which symbolic register is cheaper to spill. For comparison the number of executed

---

[4] The computation of the optimal allocation took several hours for the whole benchmark.

Benchmark	Accumulator Registers			Address Registers			
	Registers	Interferences	Copy Prop.	Registers	Interferences	Copy Prop.	Pairing
biq	46	207	8	33	77	11	2
fft	48	183	5	63	195	32	10
hdvd	34	98	9	55	176	29	5
mmult	24	61	7	43	139	28	4
vit	67	44	8	112	98	5	2

Table 2: Quantitative Statistics

Benchmark	Spills			Loads			Stores			Eliminated Copies		
	oPBQP	hPBQP	GrCo	oPBQP	hPBQP	GrCo	oPBQP	hPBQP	GrCo	oPBQP	hPBQP	GrCo
biq	2	2	2	4147	4147	4147	3424	3424	3424	3675	3675	2401
fft	4	8	7	3099	5508	4027	2903	5397	3746	4738	2955	3175
hdvd	3	3	3	2950	2950	3150	1467	1467	2667	5235	5235	2354
mmult	1	1	2	537	537	872	97	97	213	2790	2790	976
vit	0	0	0	0	0	0	0	0	0	1092	1092	728

Table 3: Qualitative Statistics

Bench-mark	Execution Time			Improvement %	
	oPBQP	hPBQP	GrCo	oPBQP - GrCo	hPBQP - GrCo
biq	157808	164977	179309	13.6	8.7
fft	80099	83399	81147	1.3	-2.7
hdvd	23370	23370	23815	1.9	1.9
mmult	8165	8165	8985	10.0	10.0
vit	194316	195671	200104	3.0	2.3

Table 4: Execution Time

loads and stores are also given in Table 3 and we can see that graph-coloring with a traditional coalescing approach performs worse than an PBQP approach. Moreover, the table contains the number of not executed copy instructions as a result of coalescing. Traditional coalescing and graph-coloring does not match the needs for register allocation in highly constraint architectures. However, the benchmark fft does not perform well in the heuristic of PBQP because a bad local spill-decision was taken due to the heuristic.

Table 4 shows the total execution time of the benchmarks. In the first improvement column (oPBQP-GrCo) we compare the execution time by using the optimal PBQP algorithm with the execution time by using the graph coloring algorithm. As given in the table the improvements range from 1.3 % to 13.6 %. The improvements by using the heuristic PBQP algorithm (hPBQP-GrCo) still performs significantly better than the graph coloring except for the fft benchmark. For the fft-benchmark the heuristic of PBQP performs poorly.

Our experiments demonstrate that our heuristic approach gives significantly better results than a traditional graph coloring approach. However, the heuristic for PBQP can be worse than the optimal allocation and there is potential for improvements in the heuristic algorithm.

7. RELATED WORK

Register allocation based on integer linear programming (ILP) was introduced by Goodwin and Wilken [7]. The approach maps the register allocation problem to an integer linear program which is solved by an NP-complete ILP-solver. The work was extended by Kong and Wilken [8] for irregular architectures. As an example they choose the IA-32 architecture and added additional features such as address mode selection. The approach can handle irregularities very nicely. However, the underlying algorithms have exponential solve time and the solvers are not able to solve bigger functions in reasonable time – they simple cut-off the solver.

Traditional graph coloring approaches [3, 2] are a full success for load/store architectures with plenty of general purpose registers and an orthogonal instruction set. However, it is very difficult to adopt the graph coloring approach for irregular architectures since it assumes that a vertex can be colored with any color as long as the interference constraints hold. This assumption is wrong for irregular architectures. Not only the color of a vertex plays an important role, also complex constraints are involved. Therfore, mapping the register allocation problem to the graph coloring problem will always be cumbersome for irregular architectures. Recently, Smith and Holloway [13] introduced an approach for irregularities based on graph-coloring. They extend the interference graph by weights and call it *weighted interference graph*. With this concept they are able to improve the colorability of aligned registers. In contrast to our approach they cannot handle register allocation properly if a symbolic register belongs to more than one register class. For example the 68K Motorola architecture has data and address registers. For some symbolic registers it is possible to store them in an address or a data register. This imprecision occurs for shared registers as well. Moreover, complex constraints between two symbolic registers such as pairing or dedicated registers are very difficult for the graph coloring approach. Previous work [2, 4, 9] have proposed some extensions for modeling irregularities. However, these extensions affect the graph-coloring heuristic and they only represent a small subset of problems encountered in irregular architectures.

Copy propagation is an important task for register allocators, which is achieved by assigning source and destination of a copy instruction the same registers. In the past several approaches as Chatin's *aggressive coalescing*[3], Briggs' conservative coalescing [2], George and Appel's iterated coalescing[6], and Park and Moon's optimistic coalescing [12] were introduced. All have in common that they coalesce vertices of the interference graph in a separate pass. The vertex selection for coalescing and the strategy of uncolored and coalesced vertices differs depending on the approach.

Our approach performs copy propagation as integral part of the reduction phase. This is necessary since irregularities can over-constrain the register assignment problem and coalescing two vertices can yield sub-optimal answers to the problem. Again, existing copy propagation approaches are primely designed for load/store architectures and they are partially applicable for irregular architectures.

So far our approach does not consider live range splitting [7], addressing modes [8], re-materialization issues [2], commutative operations [8]. Recently, an approach [1] was introduced for optimal placement of spill code for CISC architectures which can be seen as orthogonal technique for our approach.

8. CONCLUSION

Register allocation for irregular architectures demands a fundamental new approach that can cope with architectural peculiarities and has a polynomial time-complexity. Traditional approaches cannot achieve these requirements.

We have formulated the global register allocation problem for irregular architectures as partitioned boolean quadratic optimization problem (PBQP). We have shown an efficient algorithm for solving PBQP that has a nearly linear time complexity. The effectiveness and efficiency of our approach has been demonstrated with an industrial DSP C-Compiler environment.

9. REFERENCES

[1] Andrew Appel and Lal George. Optimal spilling for CISC machines with few registers. *ACM SIGPLAN Notices*, 36(5):243–253, May 2001.

[2] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):428–455, May 1994.

[3] G. J. Chaitin. Register allocation and spilling via graph coloring. *ACM SIGPLAN Notices*, 17(6):98–105, June 1982.

[4] P. Briggs K. D. Cooper and L. Torczon. Coloring register pairs. *ACM Letters on Programming Languages and Systems, (TOPLAS)*, 1(1):3–13, March 1992.

[5] G. Finke, R.E. Burkard, and F. Rendl. Quadratic assignment problems. *Annals of Discrete Mathematics*, 31:61–82, 1987.

[6] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.

[7] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocation using 0-1 integer programming. *Software & Practice and Experience*, 26(8):929–965, August 1996.

[8] T. Kong and K. D. Wilken. Precise register allocation for irregular register architectures. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-98)*, pages 297–307, Los Alamitos, November 30–December 2 1998. IEEE Computer Society.

[9] B. R. Nickerson. Graph coloring register allocation for processors with multi-register operannds. *ACM SIGPLAN 1990 Conf. on Compiler Construction, SIGPLAN Notices*, 25(6):40–52, June 1990.

[10] Homepage of Atair Software GmbH. www.atair.co.at/oce.

[11] Homepage of Infineon. www.infineon.com/dsp.

[12] Jinpyo Park and Soo-Mook Moon. Optimistic register coalescing. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 196–204, Paris, France, October 12–18, 1998. IEEE Computer Society Press.

[13] Michael D. Smith and Glenn Holloway. Graph-coloring register allocation for irregular architectures. Technical report, Harvard University, 2000.

[14] Wayne L. Winston. *Introduction to Mathematical Programming*. International Thomson Publishing, 1995.