A Retargetable Register Allocation Framework For Embedded Processors

Jean-Marc Daveau, Thomas Thery, Thierry Lepley, Miguel Santana
STMicroelectronics, Central R&D, Embedded System Technology,
850 Rue Jean Monnet,
F-38926 Crolles cedex, France
{jean-marc.daveau, thomas.thery, thierry.lepley, miguel.santana}@st.com

ABSTRACT
This paper describes the FlexCC2 register allocation framework. FlexCC2 is an optimizing retargetable C compiler for embedded processors, and in particular for DSP processors. Embedded processors often contain features such as irregular and constrained register sets that complicate register allocation, making traditional methods inefficient. In this paper, we present a register allocation framework specifically tailored for embedded processor specificities. This framework has been integrated in the FlexCC2 production compiler and is used by FlexCC2 customers.

Categories and Subject Descriptors
D.3.4 [Programming Languages]: Processors—Retargetable compilers, Optimization

General Terms
Algorithms

Keywords
Register allocation, Embedded processors

1. INTRODUCTION
Embedded processors are increasingly used in complex Systems-on-Chip (SoC) as a good compromise between flexibility and performance. This approach attempts to combine the flexibility of general-purpose programmable processors with the performance achieved by domain-specific architecture optimizations. Compilers for embedded processors must cope with these architectural optimizations and be able to exploit them.

Register allocation is a fundamental compiler component that allocates and assigns variables and temporaries to registers. The allocation phase decides which variables should reside in registers. When not all variables can be allocated in registers, some memory locations are used to store the remaining variables (spill). Once variables have been allocated to registers, the assignment phase selects a specific register for each variable residing in a register.

Embedded processors exhibit particularities that complicate these two phases compared to traditional RISC processors:

1. Embedded processors often contain a small number of registers to limit the size of the instruction word, but also to reduce processor die size and power consumption. A small number of registers complicates the allocation phase as fewer registers are available to hold variables.

2. Register sets are often irregular, meaning that not all registers are available for all instructions. Some instructions are restricted to a subset of the available registers for the source and/or destination operand. These constraints usually come from encoding constraints or an irregular datapath. Such constraints complicate the register assignment phase, as all registers are not equivalent.

3. Embedded processor instruction sets often contain instructions operating on short or long data types. Long data types are usually stored using the concatenation of two short registers. Composite registers complicate both the allocation and assignment phases.

This paper presents a register allocation framework designed to cope with the issues previously mentioned. Register allocation for embedded processors has never been addressed as such but rather as an adaptation of existing allocators. In this work we have designed a framework that specifically solves the issues brought up by embedded processors: our register allocation framework supports irregular, composite and constrained register sets.

The paper is organized as follows: section 2 presents the FlexCC2 compiler and in particular its back-end infrastructure EliXir. Section 3 list the issues raised by various non orthogonal features of embedded processors. In section 4, we detail the FlexCC2 register allocation framework and our solutions to the problems mentioned in section 3. We present some results in section 5, perspective and previous work, before concluding in sections 6, 7 and 8 respectively.
2. FLEXCC2

2.1 Architecture of FlexCC2

FlexCC2 [4] is a retargetable compiler for embedded processors, especially tailored to cope with application-specific processors and in particular AS-DSPs\(^1\). It aims at providing state-of-the-art optimizations for these processors. FlexCC2 has been designed as a modular framework allowing the development and integration of new optimizers both at high and low levels.

The architecture of FlexCC2 is represented in figure 1. It is built around the CoSy compiler development suite [1] with an in-house back-end called EliXir.

Figure 1: FlexCC2 compiler architecture

2.2 EliXir

EliXir is an STMicroelectronics proprietary back-end designed to replace CoSy back-end optimizers (i.e. the scheduler and register allocator) with a more modular infrastructure allowing more aggressive low-level optimizations. The structure of EliXir is represented in figure 2. It is based on a low-level intermediate representation built around the notion of machine instructions. The goal of EliXir is to perform analysis and optimizations that require detailed information about the target processor. Main optimizations at this level are register allocation, scheduling and peephole optimization.

Figure 2: EliXir structure

Optimizations in EliXir rely on the concept of micro-engines. A micro-engine is a low-level optimizer that can be generic or specifically designed for a given processor. Micro-engines are executed sequentially on the IR. The order in which micro-engines are chained is described in a targeting file allowing a target specific ordering of low-level optimizations.

On top of the low-level EliXir IR itself, a number of higher level APIs\(^2\) are available. These APIs are aimed at providing more complex data structures and algorithms to perform higher-level optimizations. EliXir APIs are used to write optimizing micro-engines. Among them we can name:

- The dataflow API, providing dataflow structures such as def/use, liveness.
- The structural API, which provides high-level control-flow information such as dominators, dominance frontiers and loop tree.
- The scheduling API, providing data structures for scheduling such as dependency graphs, modulo tables, reservation tables, superblocks and slot assignment. It also provides a set of local and global scheduling algorithms: linear scheduling, modulo scheduling and trace scheduling to build schedulers.
- The register allocation API, which provides interference graph construction and management, simplification and coloring. It also provides register allocation specific operations such as coalescing and spilling. This component is detailed in section 4.1.

// General purpose data registers
operation m[au4] (register base, \[0, \text{ offset}, \text{ register ax}\];
reserve = (IT, syntax "###");
register rhi = $hi$;
reserve tablo = (table, base, \{\}
reserve es(3);
reserve ar = size, (syntax "###");
read as (4);
write as (1);
write ay (3),
write ACU, use ACU, use AUX_SAR[ax];

// Multipurpose registers
register al = zhi(0), zhi(1);
register mx = std[0], std[1], zhi[1];

// Non scratch registers
num_scratch = st[2];
num_scratch = r[0], r[1], r[3];
num_scratch = r[0], r[1], r[3];

// Slots (SARU and AUX *move*)
slot SARU, SARU,
slot SARU, SARU,
write rd[0],
write rd[0],

// Functional units
unit MUL11 // DSP
unit MUL11 // DSP
unit MUL11 // DSP
unit MUL11 // DSP
unit MUL11 // DSP
unit MUL11 // DSP
unit MUL11 // DSP
declaration = (1);

Figure 3: SDF machine description

EliXir is targeted by a machine description file that describes relevant processor information for low-level optimizations: register structure, memory banks, computational hardware resources and a variety of operation features, such as output syntax, control semantics, resource usage and data flow. An example of machine description for the STMicroelectronics nandsp+ audio processor is given on figure 3.

Compared to other similar frameworks [17], the processor model handled by EliXir offers better support for irregular AS-DSPs or ASIPs architectures. Examples are composite register files, register classes, hardware loops and multiple memory spaces. For register allocation, EliXir offers numerous information on processor register sets such as:

\(^1\)Standing for Application-Specific DSP

\(^2\)Application Programming Interface
- structure of register sets/classes (contained registers, intersection, interference weight between register classes).
- structure of registers (sub-registers, parents, register inclusion/membership, register set/class membership, conflicting registers).
- scratch/non-scratch registers, colorable/non-colorable registers.

All this information is statically computed from the SDF description and stored in EliXir tables. In EliXir, register classes are used to express register constraints on operands. Typical register classes are multiplier left and right operand, low and high part of a register. Register classes can also be used intentionally to restrict the operands of a given instruction. However, register classes mostly originate from architecture or encoding constraints. An example of register class definition for the mmdsp+ processor is given in figure 4.

```
// General purpose data registers
register ch[7] width 16 ;
register r[17] width 16 ;
register rhi = rh[16] ;
register eax[7] width 16 ;
register w[18] width 16 ;

// Special registers
register x[1,3] width 16 ;

// Index registers
register xi[1,3] width 16 ;

// Intel registers
register rs[15] width 16 ;
register rt[15] width 16 ;
register rl[15] width 16 ;
register rl[15] width 16 ;

// Hardware loop registers
register rhw[15] ;

// Multiplier registers
register mi[7] width 16 ;
register ml[7] width 16 ;

// Address registers
register eax[1,3] width 16 ;
register eax[1,3] width 16 ;

// Scratch registers
register r[18] width 16 ;
```

Figure 4: Register class definition in EliXir

Register hierarchy is defined through register set concatenation. Registers resulting from concatenation are called composite registers. Simple registers are called atomic registers. Currently, EliXir supports only a flat representation of hierarchy, meaning that composite registers cannot be concatenated. Thus, composite registers are always built by concatenation of atomic registers. Each atomic register of a composite register is assigned an offset in the composite register, starting from 0 for the rightmost atomic register (see figure 5a). Figure 5a illustrates EliXir flat hierarchy. Examples of register set and classes taken from figure 4 are represented in figure 6a.

Figure 5: Register hierarchy

3. REGISTER ALLOCATION AND EMBEDDED PROCESSORS

Embedded processors often unveil characteristics, particularly on their register and instruction sets, that complicate the task of register allocation to a point of inefficiency, if not handled appropriately. In DSP applications, performance is strongly dependent on loops that must be perfectly register allocated and scheduled to meet required performance. Minor imperfections in register allocation such as bad coalescing or management of register classes can greatly decrease code quality.


The different phases of register allocation are differently impacted by embedded processor irregularities. In the following paragraphs, we give an analysis of this impact for every phase of a Chaitin-based register allocation approach.

3.1 Coloring by simplification

Coloring by simplification is driven by the degree of the coloring graph nodes. A node having a degree less than the number of colors (available registers in the target machine) is guaranteed to reside in a register (degree < K rule). On processors offering regular register files, computing the degree of a node is relatively simple. On the other hand, on a processor with both register classes and composite registers, coloring by simplification becomes harder: nodes in the graph belong to different register classes; therefore, the number of available colors varies for each node. Furthermore, the degree of a node depends on its register class, as composite registers induce variable and asymmetric degree contributions in the interference graph. An example of asymmetric degree contribution is given in figure 9 (register hierarchy is taken from figure 4).

3.2 Coalescing

Coalescing is a form of copy propagation that is performed on the interference graph. The main goal of coalescing is to eliminate redundant move operations by assigning the same color (register) to the source and the destination operands.
Move operation coalescing is an important task of the register allocator for several reasons:

- Eliminating a move reduces the total code size.
- Register constraints are expressed in FlexCC2 through move’s between register classes. As a consequence, a large number of move’s are present in the code.
- Suppressing a move may improve the result of coloring, as fewer colors are needed for a single variable.

When two coloring graph nodes are coalesced, their adjacency lists are merged, the resulting node interferences being the union of both node’s interferences. Coalescing is generally based on a coalescing heuristic which is supposed to guarantee the colorability of the coalesced nodes [5]. Classical conservative heuristics use node degree to ensure such a colorability. Applying coalescing to processors with register classes and composite registers raises two new problems related to these heuristics:

- The result of a coalescing between two registers from different classes must belong to the intersection of the source and destination register classes.
- Coalescing an atomic register with a sub-register needs to represent interferences at the sub-register level, in order to merge adjacency lists appropriately. Otherwise, the interference graph will be forced to overestimate degrees and to remove potential colors (those used by the other sub-registers of the composite register), resulting in a suboptimal coloring.

### 3.3 Color assignment

Color assignment corresponds to the last phase of Chaitin-based register allocation techniques. It is responsible for unstacking nodes from the simplification stack and assigning them a valid color (i.e. register), different from their neighbors’ colors. This phase is greatly impacted by register classes, especially when some register classes have a non-empty intersection, as illustrated by figure 6b.

In such a case, colors being part of this intersection are highly constrained. Indeed, their allocation is equivalent to removing colors from several register classes. This is even more critical when register classes having a low cardinality (such as \( m_l \) or \( m_r \) on figure 6b) are part of this intersection. Allocating such a color to a node of an unconstrained register class will prevent that color from being assigned to any of its adjacent nodes when its register class intersects with. As a consequence, low cardinality register classes may have one less color available. A potential solution would consist for unconstrained register classes to avoid using colors belonging to a non-empty intersection with constrained register classes. Unfortunately this problem cannot be solved optimally without total enumeration, resulting in over-spill: nodes are spilled when a different color assignment would have led to a lower spill solution.

### 4. THE FLEXCC2 REGISTER ALLOCATION FRAMEWORK

#### 4.1 Architecture

The FlexCC2 register allocator was designed as a framework rather than a monolithic optimization micro-engine. A divide and conquer approach was followed to solve issues one at a time. Also, our goal was to experiment several different allocation techniques without starting from scratch every time.

The register allocation framework is implemented as an EliXir API. It has been designed as a modular and flexible toolbox providing all necessary components to build an optimizing register allocator. It was designed to be extendible in all directions and is architectured as a layered software model. An existing component of the API might be extended to support new functionalities. Furthermore, components can be stacked, each new layer making use of layers below itself. Currently, two layers are available in the register allocation API (figure 7).

![Figure 7: Register allocation API](image)

The first layer corresponds to interference graph management. It implements interference graph construction, offering full support to add and remove nodes from the interference graph. The underlying representation supports composite registers and therefore all the complexity associated with such registers.

The second layer provides an implementation of the Briggs coloring algorithm with iterated coalescing [11]. This layer is supported by two components developed to improve Briggs allocation performance:

1. A spill manager in charge of optimizing spill operations through:
   - Spill location reuse to minimize space required in the stack. Spilled variables whose live range doesn’t overlap will share a memory location.
   - Spilling in unused register sets. AS-DSP processors often contains multiple and specific register sets (modulo registers, index registers, address registers) that can be used to spill variables.

2. A shuffle code manager optimizing the placement of shuffle code (move operations) inserted by the register allocator.
4.2 Main Components

The problems mentioned in section 3 have led to the development of adapted data structures and algorithms to solve them. Coalescing, and in particular register/sub-register coalescing, has driven most of the representation choices used in the FlexCC2 register allocation framework.

4.2.1 Coalescing

As discussed in 3.2, coalescing with registers of different classes and composite registers introduces very important new problems. This makes the mission of coalescing harder: guaranteeing the colorability of the coalesced nodes and propagate register class constraints.

In our framework, all possible intersections between register classes declared in the target description file are automatically computed by EliXir and made available to micro-engines through the IR API. This information includes intersections between register classes of different sizes, such as composite and atomic registers. All these additional register classes are represented in the IR and available as any register class declared by the compiler retargeter. The availability of this information is of great help for register allocation. Moreover, any empty intersection is reported by EliXir, as it may lead to an impossible coalescing.

Register/sub-register coalescing is the hardest issue to solve in coalescing. Unfortunately, this is a frequent case in DSP’s multiply-accumulate loops, where single and double precision registers are combined to fulfill application precision requirements. Such a coalescing needs to represent that only a part of a composite register is alive in the interference graph. Reusing the free part of such registers may be determinant for performances of processors with few registers. Adding this information to the interference graph leads to several consequences:

- Interference graphs are usually represented as an adjacency matrix plus an adjacency list for efficiency reasons. The matrix elements are boolean values corresponding to the presence or absence of interference. Such boolean values are not suitable for sub-register interferences (see 4.2.2).

- Liveness of composite registers must be computed on a sub-register basis. A composite register is no longer considered as a single entity within this computation but is instead decomposed into its atomic sub-registers. This may have a big advantage: any potential register hierarchy (e.g. single/double/quad) has no impact in liveness computation, greatly simplifying it.

4.2.2 Interference graph representation

Our framework extends adjacency matrices in order to support composite registers: boolean values used to represent interference values are replaced by a bitfield, where each bit corresponds to an atomic register of the composite register. An atomic register is represented by the bit $i$, where $i$ corresponds to its offset in the composite register.

This representation allows computing contribution to degree and interferences at the atomic register level, i.e., the lowest granularity and highest precision available, independently of any register hierarchy. On the other hand, as register sets or classes may have different sizes, their interference matrices may become asymmetric as interference relations become asymmetric. Using this approach, an atomic register may only interact with sub-registers of a composite register. This property allows the register allocator to reuse dead parts of composite registers.

Using this representation, the complexity of the Briggs algorithm remains unchanged at $O(n^2)$. The overhead induced by our bitfield matrix appears with offset manipulation that occurs during interference graph maintenance, coalescing or color assignment.

![Bitfield interference representation](image)

Figure 8: Bitfield interference representation

4.2.3 Computing degrees of interference graph nodes

Node degrees must be computed as precisely as possible because they drive two main phases in the graph coloring stage: simplification and coalescing.

The selected interference graph representation simplifies the accurate computation of the degree of each node. However, we have added a specific data structure to compute degrees resulting from interactions with other (composite) registers. This data structure is a matrix storing the degree contribution of a register class to other register classes. This matrix might be asymmetric, as register classes of different sizes have different degree contributions. Such a matrix is statically computed from the register sets and classes available in the EliXir IR (explicitly declared or resulting from an intersection).

An example of degree a contribution matrix is given in table 1. It can be noted that asymmetry is introduced by register sets of different sizes. For instance, the sub-matrix for register sets $r_{lh}$, $r_h$, $hwlr$, $m_r$, $m_l$ is symmetric because they share the same size. Figure 9 illustrates the degree contribution between register sets of different sizes.

![Degree contribution matrix](image)

Table 1: Degree contribution matrix

Figure 9a & 9b show an asymmetric degree contribution between a single register $r_{lh}$ and a double register $r$. From the $r_{lh}$ side, an $r$ register consumes both an $r_l$ (low part of $r = r_{lh}$) and an $r_h$ (high part of $r = r_{lh}$), removing two registers (colors) from $r_{lh}$ choice. The contribution degree
of $r$ on $r_{lh}$ is therefore two. From the $r$ side, an $r_{lh}$ register will consume either the low part or the high part of an $r$ register, therefore removing only one register from the $r$ choice. The degree contribution of $r_{lh}$ on $r$ is then one. If we consider an interference between an $r_1$ register and an $r$ register, as represented in figure 9c & 9d, then from the $r_1$ side, the $r$ register consumes only an $r_0$ (= $r_1$) register as $r_1 \cap r_0 = \emptyset$. The degree contribution of $r$ on $r_1$ is one and reciprocally. When considering an interference between a single register ($r_0$) and a sub-register of a composite register $r$, we compute the contribution using the register class of the concerned sub-register of $r$ ($r_0 \subset r_1$ for example) and the degree contribution is symmetric and equal to one. In this case, we must be conservative and assume that the $r_{lh}$ register will be allocated in an $r_1$, therefore conflicting with $r_0$. The corresponding bitfield interference matrix for figure 9 is represented in figure 10.

![Figure 9: Degree contribution asymmetry](image)

![Figure 10: Bitfield interference matrix](image)

### 4.2.4 Simplification

Coloring by simplification is driven by the rule: $degree < K$. This rule is adapted in our framework in order to support multiple register classes: $K$ is no longer considered as a constant during graph simplification, but rather as a variable of the node’s register class. For each register class, the number of available colors $K$ is equal to the cardinality of that class (number of its registers), as illustrated by table 2. For all possible register class intersections (of register classes defined in the SDF target description), EliXir computes the corresponding $K$. Table 3 represents some of them (empty and non-empty). Only colorable registers are taken into account. Non colorable registers (e.g. a general purpose register used as a stack pointer) are ignored. These rules allow us to offer the same colorability guarantee as in [5] and [18].

![Table 2: Register class $K$ for $K$-coloring rule](image)

#### 4.2.5 Color assignment

Color assignment is solved heuristically and conservatively in FlexCC2, in order to cope with register classes. The solution is based on the rule $degree < K$ completed with an optimistic spill [7], ensuring the colorability of simplified nodes and maximizing the chances of spilled nodes to receive a color. During simplification, a heuristic chooses the node to simplify among all simplifiable nodes obeying to the $degree < K$ rule, giving the lowest priority to register classes with low cardinality. In other words, such nodes end up on top of the coloring stack and therefore receive a color first, maximizing their chances to receive a valid color. For instance, nodes from register classes $m_l$, $m_h$, and $hwl$ in figure 6 are simplified at the end whenever possible.

Our color assignment heuristic avoids using colors belonging to low cardinality register classes. For instance, as shown in figure 6, it will avoid using colors $r_{16}$, $r_{14}$, $r_{12}$ (resp. $r_{10}$, $r_{8}$) for register class $r_{16}$, as these colors also belong to the low cardinality register classes $m_l$ (resp. $m_h$). Optimistic spill [7] has a beneficial impact for register classes with low cardinality. Indeed these classes, being in general good candidates for frequent spills, are favored by this technique, which maximizes their chances of receiving a color.

### 4.3 Retargeting

The register allocation framework is based on processor information provided by EliXir, and so it is retargeted via EliXir. The only target specific part of the register allocator is a library provided by the compiler writer to generate specific code sequences for spilling, register save and restore, register moves and stack pointer updating. These functions ensure the correct behavior of the register allocator for a given target, completing the framework retargeting.

### 5. RESULTS

The FlexCC2 register allocator framework has been integrated in the FlexCC2 production compiler for the STM-mmdsp+ single-mac audio processor. Results for version 2.23 of the compiler are given below. Results are given for two architectures of the mmdsp+ processor, $op9$ (16 bits DCU/16 bits ACU) and $ha$ (16 bits DCU/24
bits ACU), using the -O4 compilation option. Table 4 gives the results for the EFR (Enhanced Full Rate GSM codec) benchmark, tables 5 and 6 for the G723.1 and AMR audio codecs respectively. Table 7 shows the results on the EFR codec using the -O3 compilation option, therefore releasing register pressure in loops. Results encompass the number of coalesced moves, the number of spilled variables, the mips\(^3\) computing power required to execute the application in real time\(^4\), the code size in 64 bits VLIW instruction words\(^5\), and the allocator running time in seconds. They are given for the boolean interference matrix/symmetric degree contribution matrix and the bitfield interference matrix/asymmetric degree contribution matrix (table 1) cases.

In all benchmarks, we observe a significant reduction of spill (9% to 15%) and an increase in the number of suppressed moves (12% to 19%) when using our bitfield interference/asymmetric degree contribution matrices, resulting in a small performance and code size gain. The small impact on performance (1.6% to 5%) can be understood as most of the performance requirements are in loops that are already nearly perfect using the aggressive -O4 compilation option. However, a 2% gain in performance on the EFR is significant considering that, at 17.83 mips\(^6\), we are close to the theoretical limit of the mmdsp+ processor for this application (\(\approx 15\) mips). The impact on code size is also largely hidden by the available instruction word parallelism. Through more precise interferences, the degree of nodes is more accurate and leads to better coloring and improved coalescing. Indirectly, better register allocation opens more opportunities for improved scheduling and software pipelining. The allocator running time overhead ranges from 0% to 32%, being above 15% in only 3 cases. Considering the number of registers to allocate for each application (\(\approx 5300\) for EFR, \(\approx 4600\) for G723.1 and \(\approx 14600\) for AMR), the actual overhead is moderate (\(< 1.5s\)) in most cases. It exceeds five seconds in only 2 cases.

### Table 3: \(K\) for various register class intersection

<table>
<thead>
<tr>
<th>node</th>
<th>(r \cap r_{lh})</th>
<th>(r \cap r_{hi})</th>
<th>(r \cap r_{lhi})</th>
<th>(r \cap m_r)</th>
<th>(m_r \cap m_r)</th>
<th>(m_r \cap m_{lh})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r)</td>
<td>{0}</td>
<td>{0}</td>
<td>{0}</td>
<td>(r_{hi}, r_{hi})</td>
<td>(r_{hi})</td>
<td>(r_{hi} \cap r_{hi}}</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

\(\text{Table 3: } K\text{ for various register class intersection}\)

EliXir uses a flat representation for register hierarchy, resulting in some limitations for coalescing. We illustrate this through an example based on figure 5: only atomic registers \(ext, r_i, r_h\) exist in \(x_r\) for EliXir and absolutely not \(r\); thus coalescing between \(x_r, 0\) and \(r\) is not possible because it cannot be expressed in EliXir semantics, where \(x_r, 0\) corresponds to an \(r_i\) register and \(n\) (see figures 5a and 5b). This limitation definitely prevents the register allocator from performing coalescing on MMX-like registers sets organized following a quad/double/single hierarchy style. Thus, we need to extend EliXir semantics to fully support hierarchies, introducing the sub-register concept represented in figure 5b. The flattened representation used by EliXir will not be affected by this extension, and so the atomic register concept will continue to be available.

Currently, a third layer in the register allocation framework is under development: a fusion-based register allocator [16]. This layer will offer integrated live range splitting and spilling, allowing a variable to be spilled only in regions of high register pressure. This capability will offer a great advantage over the Briggs heuristics which spill a variable for its whole live range. We adopted the fusion approach over [10, 3], because we feel it will allow us to build various hierarchical register allocation algorithms, offering more opportunities for complex allocation approaches to be built on top of it. We plan, in particular, loop-oriented allocation algorithms.

### 7. PREVIOUS WORK

Register allocation has been widely addressed in literature, and many approaches have been proposed. Most of them are related to Chaitin’s [9] approach, but very few address problems raised by embedded processor’s architecture:

- Support for irregular registers via register classes and register set concatenation. Although [15] proposed an approach using integer-programming supporting irregular register sets, this approach requires modeling of register constraints by inequations, making it difficult to implement in an industrial compiler. None of this is necessary in FlexCC2, where only a natural register definition is required (see figure 4). Thus, register structures and constraints are easily modeled in the SDF target description, and automatically included in the EliXir IR for register allocation.


- Constrained register sets. Embedded processors are often characterized by a small number of registers. Algorithms and heuristics have been adapted in our infrastructure to limit spill when few registers are available.

---

\(^3\) Million Instructions Per Second

\(^4\) As the mmdsp+ executes 1 VLIW instruction/clock cycle (\(cpi = 1\)) this number is also the number of clock cycles required to execute the application (1 mips \(\Rightarrow 1\) million clock cycles).

\(^5\) An mmdsp+ processor delivering 1 mips/MHz will have to run at 17.83Mhz to execute the EFR in real time.
We have paid a lot of attention to the degree computation in the presence of register classes and register hierarchy. To our knowledge, this has not been addressed in the literature.

- Our register allocation framework was designed to be open and extendible, allowing different techniques to be implemented on top of it. Currently, two register allocator micro-engines have been built on our framework: a standard Briggs allocator [5] and a Callahan [8] hierarchical allocator. Moreover, a new layer is being developed to allow live range splitting register allocation [16]. We know no other framework that offers such possibilities for register allocation.

[18] presents a generalization of the degree < K test, called the (p, q) test, to handle irregular register sets and register classes. The degree < K approach with asymmetric degree contribution, presented in this paper, offers the same colorability guarantee. In their approach, registers are annotated with their register class, allowing the selection of the appropriate p for the (p, q) test. Coalescing, degree computation, sub register and register class intersection computation are not addressed.

VPO [12] is a portable optimizer for DSPs, based on the Zephyr retargetable compiler infrastructure [2]. VPO address the problem of heterogeneous register classes by computing and propagating along the IR tree the correct reduced register class (register class resulting from the intersection of the target/source operand of two instructions) that allows registers to be allocated without inserting an extra move operation. In EliXir, this step is performed during code generation by inserting move between operands of operations to express registers class restrictions on operands. These move are later removed by the conservative coalescing algorithm [11] of the register allocator. Compared to our approach, this is equivalent to performing aggressive coalescing, which can cause over-spill.

The PROPAN postpass optimizer [13] relies on integer-linear-programming to perform register allocation and scheduling. PROPAN does not support explicitly register class as in EliXir and does not express constraints on operand using move. In TDL [14], expressions describe relationships between resource usage, operands and scheduling properties of machine operations. Their formalism allows the specification of parallel execution of operations, operation sequencing and resource restrictions on operands that are allowed. Expressions specified in TDL are transformed into constraints for the linear solver. In EliXir, parallelism is expressed through reservation tables and constraints on operands through register classes. Constraints such that if two operations are to be scheduled in the same control step, their operands must reside in the correct register group cannot be expressed explicitly in SDF, while in PROPAN, such constraints can be expressed directly in TDL.

8. CONCLUSION

In this paper, we have described a retargetable and extendible register allocation framework for embedded processors. Although this framework implements classical register allocation heuristics, it differentiates itself from other approaches on many points: it is easily retargetable, well-adapted to embedded processors, it solves issues such as register/sub-register coalescing and color assignment in presence of register classes, and it is highly efficient on processors with few registers. It is integrated in a coherent back-end framework consisting of a powerful low-level intermediate representation and a set of optimizers. Last but not least, it has proven its efficiency in an industrial compiler with extremely satisfying results.

Acknowledgments

The authors wish to thank all the members of the FlexCC2 team for their grateful support during the development of the register allocation framework: Claire Robine, Frederic Riss, Denis Pilat and Valerie Bertin. We thank all the anonymous reviewers for insightful comments on the paper and Jean-Claude Bauer for his corrections.

9. REFERENCES

Table 6: AMR results

<table>
<thead>
<tr>
<th></th>
<th>ha architecture</th>
<th>op9 architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>move</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>spill</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>mips</td>
<td>higher</td>
<td>lower</td>
</tr>
<tr>
<td>size</td>
<td>lower</td>
<td>higher</td>
</tr>
<tr>
<td>time</td>
<td>faster</td>
<td>slower</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>reduce</th>
<th>increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean/symmetric</td>
<td>+12%</td>
<td>-9.2%</td>
</tr>
<tr>
<td>bitfield/asymmetric</td>
<td>-2.2%</td>
<td>-1.3%</td>
</tr>
<tr>
<td></td>
<td>+1.9%</td>
<td>+12%</td>
</tr>
<tr>
<td></td>
<td>-13%</td>
<td>-2%</td>
</tr>
<tr>
<td></td>
<td>-1.2%</td>
<td>+15%</td>
</tr>
</tbody>
</table>

Table 7: EFR results (-O3)

<table>
<thead>
<tr>
<th></th>
<th>ha architecture</th>
<th>op9 architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>move</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>spill</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>mips</td>
<td>lower</td>
<td>higher</td>
</tr>
<tr>
<td>size</td>
<td>lower</td>
<td>higher</td>
</tr>
<tr>
<td>time</td>
<td>slower</td>
<td>faster</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>reduce</th>
<th>increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean/symmetric</td>
<td>+15%</td>
<td>-12%</td>
</tr>
<tr>
<td>bitfield/asymmetric</td>
<td>-1.6%</td>
<td>-1.4%</td>
</tr>
<tr>
<td></td>
<td>+7.7%</td>
<td>+17%</td>
</tr>
<tr>
<td></td>
<td>-15%</td>
<td>-1.7%</td>
</tr>
<tr>
<td></td>
<td>-1.4%</td>
<td>+23%</td>
</tr>
</tbody>
</table>


