The Structure of a Forth Native Code Compiler*

M. Anton Ertl Christian Pirker
Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8, A-1040 Wien
{anton,pirky}@mips.complang.tuwien.ac.at
http://www.complang.tuwien.ac.at/projects/forth.html
Tel.: (+43-1) 58801 4474
Fax.: (+43-1) 505 78 38

Abstract

Writing a sophisticated Forth native code compiler poses some tasks that are not discussed in compiler text books. Some of these tasks arise from specific language features, others from the requirement for very fast compilation to maintain interactivity. In this paper we describe some of the more interesting data structures and algorithms used in the RAFTS prototype.

1 Introduction

In [Ert92, EP96] we discussed many engineering aspects of a RAFTS compiler only superficially, if at all. In the meantime we have also made some changes in preparation for interprocedural optimizations. In this paper we discuss the implementation issues of our RAFTS prototype compiler in more depth.

Many of the issues discussed here are due to specific properties of the language Forth:

Interactivity demands very fast compilation. This led to the use of a fast instruction selection technique (Section 4.2), an improved instruction scheduler (Section 4.3), simple local register allocation (Section 4.4), and the integration of all these phases into a few passes (Section 4.6).

High call frequency requires inlining and/or interprocedural register allocation for good performance. This means the compiler should not generate code for a word when its source code is read. Instead, it should wait as long as possible, to make use of information about the word's callers. This leads to an intermediate representation for words that were not compiled yet (Section 3).

Mixing compilation and execution means that the compiler cannot wait for the whole program to be read, and generate code then, but may have to execute a part of the program before everything is read. Together with the requirement to compile as late as possible, this has led to implementing code generation on `EXECUTE` (Section 3.2).

Stratification of definitions (i.e., define before use) is required in Forth and gives the compiler opportunities for interprocedural optimizations whose exploitation is more difficult or impossible in non-stratified languages and languages with separate compilation. We do not discuss these optimizations here, but we have prepared for them (Section 3).

Instead of repeating the Motivation and Related Works sections from [EP96], we suggest reading it.[1]

2 Compilation actions

In [EP96], we discussed various alternatives for dealing with the problem that native code compilers require an unusual and nonuniform implementation of compilation. At that time, our prototype compiler used a combination of the *wordlist* approach (i.e., a wordlist for the compilation action and a wordlist for the interpretation action) and the *intelligent `COMPILE,`* approach (i.e., `COMPILE,` performs a special compilation action for each execution token), where the wordlist part caused non-standard behaviour.

In the meantime we have switched to a pure intelligent `COMPILE,` approach. The challenge was doing this efficiently while maintaining the integration with the Gforth threaded code system.

Threaded code systems provide information in the execution token (XT) on how to `EXECUTE` the definition represented by the execution token, but

*This work is supported by the FWF, research project P11231-MAT.

[1] Our papers are available through http://www.complang.tuwien.ac.at/papers/.

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in: Euroforth '97 Conference Proceedings, pages 107-116

`xt-cfa`	code field
`xt-defer`	CFA for deferred definition
`xt-compile`	XT of compilation action
`...`	other fields
`xt-pfa`	parameter field (body)

Figure 1: Structure of a native code definition

no information on how to `COMPILE,` it. The reason is that all XTs are `COMPILE,`d in a uniform way, with `,`.

Native code compilers, on the other hand, often want to do something different depending on the XT that is `COMPILE,`d[2]; e.g., our prototype `COMPILE,`s the XT of `+` with a word called `compile-+` (which is obviously inappropriate for `COMPILE,`ing other words).

One solution would be to keep the data structures of the threaded code system and perform the different `COMPILE,` actions for different XTs by performing a table lookup in `COMPILE,`. Another solution is to change the data structure for a definition to include information on the compilation action. We chose the latter approach (see Fig. 1).

If the threaded code system and the native code compiler have different data structures for definitions, how can they work together? There are two directions to consider:

- Passing the XT of a native code definition to a word expecting a threaded code XT. This presents no problem, because we designed the data struture of native code definitions as extension of the threaded code data structure (see Fig. 1): The XT can still be used as a code field address (CFA).
- Passing the XT of a threaded code definition to a word that expects a native code XT (and wants to `COMPILE,` it). This would cause a crash, and must be avoided.

So, how can we use threaded code words in native code? We decided to create aliases for the threaded code words we use; these aliases have a native code definition structure. The code field for such an alias (used for `EXECUTE`) contains `dodefer`, the `xt-defer` field contains the CFA of the threaded code word, and the `xt-compile` field (used for `COMPILE,`, contains the XT of `compile-threaded`).

The default search order contains only wordlists consisting only of native code structured definitions. We explicitly create native code aliases in one of these wordlists for all the threaded code words we use. This is not very elegant, but simple, and has further advantages: In particular, we cannot accidentially invoke a threaded code word that tries to compile something with `,`.

[2]Note that this issue is independent of the question of non-default compilation semantics: `COMPILE,` must compile the execution semantics given by the XT, not any other semantics; native code compilers just want to perform an XT-dependent action when compiling the XT, because native code is not as uniform as threaded code, and to allow optimizations.

3 Delaying code generation

For interprocedural register allocation we need to compile as many words together as possible. For inlining decisions, we need to know how often a word is called before inlining it. In both cases we have to delay the code generation as long as possible.

3.1 Linear intermediate representation

Of course, we have to represent the program in some data structure between the time when the program is compiled (as perceived by the user), and the time when the code is generated for it.

An obvious candidate would be the data flow graphs that are already used in code generation for basic blocks. The control flow would have to be added somehow. However, this representation is not only quite space-consuming, it also complicates code replicating optimizations like inlining: copying a graph is somewhat complex; copying a control-flow/data-flow graph hybrid and inserting it in another such beast would be worse.

A linear representation of the graph would be nicer. A stack-based representation is easy to transform into a graph, and it is easy to create from the source code. So, we settled on a stack-based code that is quite close to the source. The main difference from source is that the names are resolved. Another difference is that the program is not represented as strings, but as threaded code; thus, when we finally want to generate executable code for a piece of intermediate representation, we just have to execute the intermediate representation; if we want to inline a colon definition, we just have to call its intermediate representation while compiling its caller (instead of producing separate code for it, and generating code for a call to that code).

While compiling source code into the intermediate representation, the compiler already collects data about the stack effects and other information that will be useful in interprocedural register allocation and other interprocedural optimizations.

3.2 Compile on `EXECUTE`

We want to generate code for a definition at the latest possible moment, and that moment is when the definition is `EXECUTE`d (e.g., by interpreting

`il-left`	left child (or unused)
`il-right`	right child (or unused)
`il-op`	node type (operator)
`il-val`	value (of `I_LIT` node)
`il-reg`	target register
`il-slabel`	state (for inst. selection)
`il-nt-insts`	ML nodes for this IL node
`il-depends`	dependences (for scheduling)

Figure 2: IL node fields

the word). At that moment, the compiler generates code for the `EXECUTE`d definition and all the definitions that this definition calls statically (i.e., without `EXECUTE` etc.). This technique of dynamic translation was pioneered by [DS84].

This is achieved by putting a `docodegen` code address in the code field (`xt-cfa`) of all definitions for which no code has been generated yet. When the definition is `EXECUTE`d, this routine is invoked: It generates code for the definition (by executing the intermediate representation) and all its descendents in the static call graph. For all definitions, for which it generates code, it replaces the code address `docodegen` in the code field with a `docode`, the code address for executing native code definitions. After all the necessary code is generated, it finally executes the native code for the definition.

3.3 Interprocedural optimizations

We are currently working on adding interprocedural register allocation and intend to add inlining; since we have not completed this work, we do not discuss it in depth here.

These optimizations will collect data while the linear intermediate representation is built, and possibly in separate passes just before code generation (i.e., at the start of `docodegen`). Then the compiler makes inlining decisions and determines the register allocations at the basic block boundaries. These decisions are later used during code generation.

4 Basic Blocks

4.1 Data Structures

We use two data structures during code generation for basic blocks: The IL (intermediate language) data flow graph is a (mostly) machine-independent representation of the basic block; the ML (machine language) data flow graph consists of nodes representing the target machine instructions.

We have described the construction of the IL graph for a basic block in [EP96].

Figure 2 shows the fields of an IL graph node. The `il-op` field contains the node type (or operator) and is discussed below. If the result of the operation should reside in a special register (say, the return stack pointer register), the `il-reg` field contains the register number. The other fields are explained in later sections.

Type	Operands
`I_INVERT`	1
`I_NEGATE`	1
`I_PLUS`	2
`I_MINUS`	2
`I_TIMES`	2
`I_SLASH`	2
`I_MOD`	2
`I_AND`	2
`I_OR`	2
`I_XOR`	2
`I_LSHIFT`	2
`I_RSHIFT`	2
`I_SRSHIFT`	2
`I_CFETCH`	1
`I_FETCH`	1
`I_CSTORE`	2
`I_STORE`	2
`I_EQUALS`	2
`I_LESS`	2
`I_ULESS`	2
`I_0BRANCH`	1
`I_BRANCH`	0
`I_CALL`	0
`I_LIT`	0
`I_ZERO`	0
`I_MOVE`	1
`I_REG`	0

Figure 3: IL node types

Figure 3 shows the node types (or operators) of the graph. Most correspond directly to some Forth word, and their names are pretty self-explanatory. All other Forth words are built from these primitives or call threaded code words. `I_RSHIFT` is the unsigned (logical) shift right, `I_SRSHIFT` is the signed (arithmetic) shift right; `I_REG` represents a specific register (e.g., a stack pointer register) in the data flow graph (`il-reg` contains the register number); `I_MOVE` copies a register and is needed for dealing with register shuffling (see Section 4.4.1).

The IL operators discussed above are machine-independent. In addition, we have an operator `I_LITS` that represents a small (15 bit) constant and is useful for the MIPS architecture, which can encode such constants directly in instructions. For other architectures there would be other such operators. Dealing with this problem already at the IL level makes retargeting a little more complex, but allows more accurate decisions in the scheduler and register allocator, and simplifies the assembler

`ml-left`	left child (or unused)
`ml-right`	right child (or unused)
`ml-asm`	XT for code emission
`ml-val`	value of a literal field (if any)
`ml-reg`	destination register
`ml-count`	reference count (for scheduling)
`ml-depends`	dependences (for scheduling)
`ml-latency`	used for scheduling
`ml-let`	latest ex. time (for scheduling)
`ml-pathlength`	used for scheduling

Figure 4: ML node fields

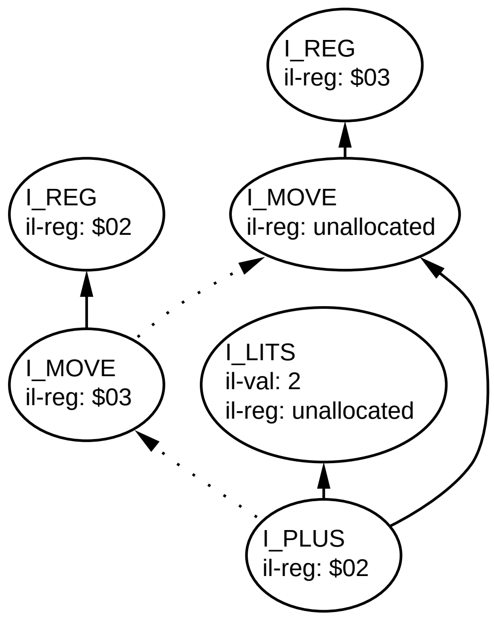

Figure 5: The IL graph for `swap 2 +`

(the other logical place to deal with overly large constants).

The ML graph is built during code selection. Each node in the ML graph corresponds to (usually) one instruction. Figure 4 shows the fields of an ML graph node.

`Ml-asm` contains the execution token (XT) of a word that actually creates the machine code later, during code emission (see Section 4.5). `Ml-val` and `ml-reg` are also needed for code emission.

Figure 5 shows the IL graph for `swap 2 +`, assuming that the top-of-stack item is in register `$02` and the second stack item is in register `$03`, both on entry and exit of the basic block; Fig. 6 shows the ML graph, and Fig. 7 shows the resulting machine code. The reasons why the graphs look like this and why the fields contain what they contain are discussed in the following sections; e.g., Section 4.4.1 explains the presence of the `I_MOVE` nodes.

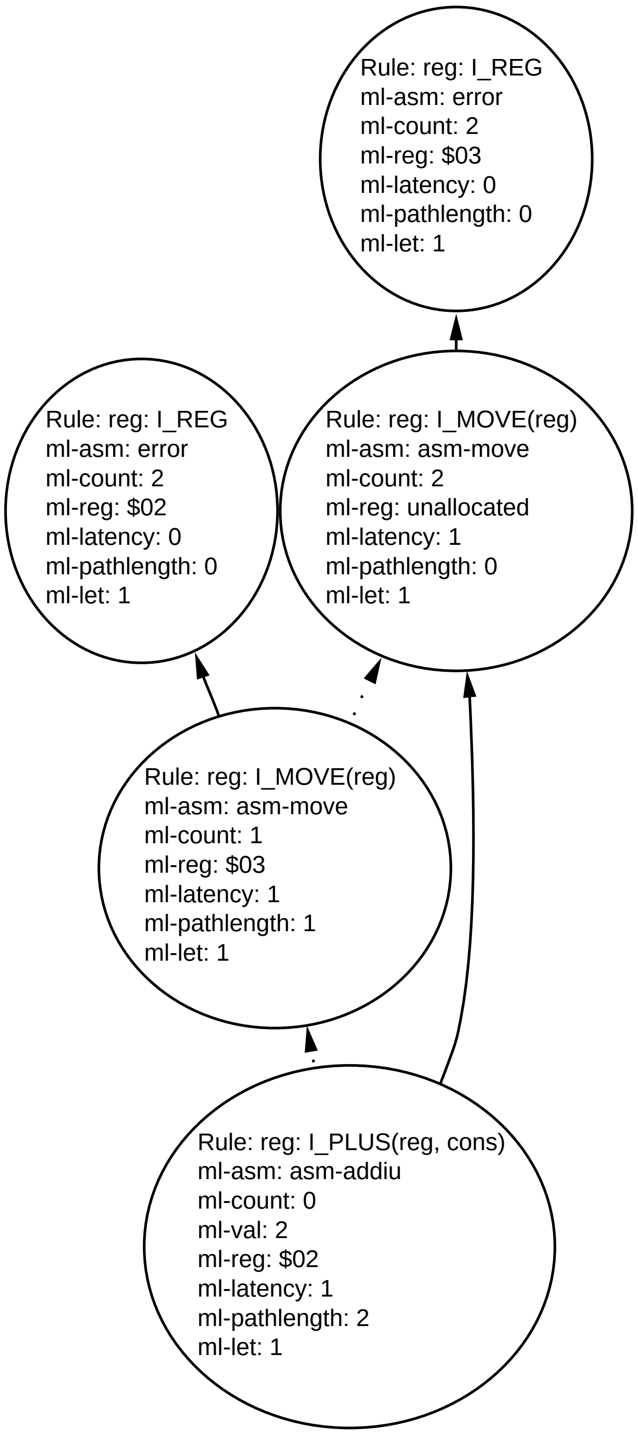

Figure 6: The ML graph for `swap 2 +`, before scheduling and register allocation

```
move $4,$3
move $3,$2
addiu $2,$4,2
```

Figure 7: Machine code generated for `swap 2 +` (the destination register is on the left)

4.2 Instruction Selection

4.2.1 Tree Parsing

The method we use for instruction selection is called tree parsing. We use it because such instruction selectors are very fast, and because it is easy to use (with the right tools).

Basically, for a given IL graph the instruction selector produces the equivalent ML graph. We describe the equivalence in rules, such as:

```
reg:I_PLUS(reg,cons)#1#binrc-inst#asm-addiu
```

Each rule contains four parts, separated by `#`:

1. The first parts of all rules constitute a tree grammar for parsing IL trees; in this example, it says that we can parse the tree as a `reg` with this rule, if the operator is `I_PLUS`, the subtree rooted at its left child can be parsed as `reg`, and the subtree rooted at its right child can be parsed as `cons`. `reg` and `cons` are called *nonterminals*; instead of *parse as `reg`*, we also say *reduce to `reg`*. In the context of our code selector, this means that we have an instruction for the addition of a register and a (small) constant, delivering the result in a register.

2. The cost of using this rule to parse the tree; the cost should reflect the cost (e.g., code size, number of cycles) of the instruction that is generated for this rule. If there are several ways to parse the tree, the tree parser chooses the cheapest. The cost of the instruction above is 1 cycle.

3. The action, i.e., a piece of Forth code for building the ML node for the instruction described in the rule (if such a node is built for the rule). In this example, a word for building a node for an instruction with a register and a constant operand.

4. A piece of Forth code for doing the code emission for the ML code. This piece of code is wrapped in a colon definition, and its execution token is passed to the ML node building code (part 3). This XT is then stored into the field `ml-asm`. In this example, a piece of code for generating an `addiu` instruction.

We use the tree parser generator Burg [FHP92, Pro95] for translating these rules into a tree parsing automaton. We have modified Burg to generate Forth source code (instead of C) for this purpose.

The tree parsers generated by Burg work in two passes over the tree:

- The *labeller* runs from the leaves to the root and computes a *state label* for each node (in `il-slabel`), based on the node's operator, the states of its children, and a table generated by Burg.

- The *reducer* computes the rules used for parsing the tree, starting at the root. Our compiler reduces the complete tree to `reg` (the goal nonterminal). The reducer determines the rule used from the state label and from the goal nonterminal. It then reduces the subtrees corresponding to the nonterminals on the right-hand side of the rule to these nonterminals; e.g., if the example rule above is used, it reduces the subtree rooted at the left child to `reg`, and the subtree rooted at the right side to `cons`.

  Since just knowing the rules used is not very useful, the reducer typically also performs some code generation action; in our case it builds the ML graph, usually one ML node per rule application.

  Note that the reducer may visit some IL nodes several times; this happens for rules like

  ```
  reg:cons#...
  ```

  where the IL node is first visited for the goal nonterminal `rge`, and then for the goal nonterminal `cons`.

  The reducer may also skip some IL nodes; this would happen with a rule like

  ```
  reg: I_FETCH(I_PLUS(reg, cons))#...
  ```

  which would skip the `I_PLUS` node. Therefore, the reducer can translate one IL node into several ML nodes, but also several IL nodes into only one ML node.

4.2.2 The Reducer

The labeller is quite simple. However, the reducer merits further discussion: Basically, it performs a recursive walk of a tree of rules that were used for parsing the IL tree. On returning from reducing the subtrees, the reducer performs the action (part 3) of the rule, passing the following items on the data stack: the stack items produced by the actions of the subtrees, the XT for part 4 of the rule, the address of the root IL node of the tree, and the goal nonterminal. Typically, the action consumes these items and produces some new items.

Usually, the action builds an ML node, that contains the ML node pointers passed in from the subtree reductions as children (`ml-left` and `ml-right`), and the XT for part 4 in `ml-asm`; the action then returns a pointer to that ML node.

You may wonder how we keep the stack balanced; after all, we don't know which rules are used for

`reg`	-- ml	inst. with normal register result
`con`	-- n	constant
`cons`	-- n	small constant
`addr`	-- ml n	effective address

Figure 8: Nonterminals and their reduction stack effects (for the MIPS architecture)

reducing the subtrees, and, therefore don't know the actions used. What we do know, however, is, to which nonterminals the subtrees are reduced. We therefore make sure that all rules that reduce to the same nonterminal have the same overall stack effect; this overall stack effect consists of the stack effects for reducing the subtrees, then pushing three stack items, and then the action of the rule.

Figure 8 shows the nonterminals that we currently use in the MIPS code selector and their reduction stack effects. The most important one is `reg`; reducing an IL tree to a `reg` creates an ML tree for instructions that compute the expression represented by the IL tree and leave the result in a register. The ML node pointer returned by reducing to `reg` is the root of this ML tree. For simplicity, we also use this nonterminal for instructions that do not have a result in a register, like `I_STORE` and `I_BRANCH`. Trees consisting of constant computations can be reduced to `con` and, sometimes, `cons`; the reduction returns the value of the tree. The `addr` nonterminal is a factor of the memory access instructions; an effective address on the MIPS architecture is the sum of a register and a small constant; reducing a tree to an `addr` simply returns the constant and an ML node pointer for a tree that computes the register.

4.2.3 DAGs

You may have noticed that we have discussed trees in the above, but use the tree parser on directed acyclic graphs (i.e., the IL graph); this is possible, with a little care.

One of the effects of applying tree parsing to a DAG is that the reducer may visit the same node several times, from different parents. Moreover, it may try to reduce the tree rooted at that node to different nonterminals. What we want the code selector to do when encountering an IL node with more than one parent (a shared node) is to create a shared ML node.

The `il-nt-insts` field of the IL node plays an important role here: It contains an array indexed with nonterminal numbers; each element of the array stores a pointer to the root of the ML graph that is built by reducing the IL graph rooted at the node to the respective nonterminal; it contains 0, if such an ML graph has not been built (yet). For our MIPS code selector, only the `reg` elements contain such pointers (because ML nodes are only built when reducing to a `reg`).

Now, before reducing an IL graph to a nonterminal, the reducer checks the `il-nt-insts` element for the nonterminal. If it already contains an ML node pointer, then the reducer just returns that pointer instead of reducing the IL graph again; if it contains 0, then the reducer performs the reduction as usual, and the actions that create ML nodes put them in the appropriate `il-nt-insts` element.

This shares the ML nodes and avoids replication in cases where all parents try to reduce to the same nonterminal. For the other cases, we just write the rules in a way that avoids unprofitable code replication. We will discuss this (very involved) topic somewhere else.

4.3 Instruction Scheduling

The instruction scheduler determines the execution order of the instructions. The result of the scheduler is an array of ML nodes.

4.3.1 Dependences

The scheduler has to consider more dependences between the instructions than the register data flow dependences represented by the `ml-left` and `ml-right` fields. E.g., it has to take into account write-after-read dependences through registers and dependences through memory. These additional dependences are represented in the IL graph by the `il-depends` field, which contains a list of pointers to IL nodes on which the instruction depends. In Section 4.2 we skipped the problem of how to translate these dependences into dependences in the ML graph; therefore, we will discuss this problem here.

In dealing with this problem, we exploit two facts:

- The dependence pointers only point to nodes with the operators `I_FETCH`, `I_STORE`, `I_CFETCH`, `I_CSTORE` (for memory dependences) and `I_MOVE` (for register write-after-read dependences).
- All rules involving these operators are of the form

  ```
  reg: operator(...)#...
  ```

  where *operator* is the operator in question, and it only occurs in this place.

Now, we extend the reducer in the following way: During its recursive walk of the graph, it now also follows the dependences in `il-depends`, and reduces the subtrees rooted there to `reg`. It then adds the resulting ML node pointer to the list of dependences in `ml-depends`.

4.3.2 Scheduling

For the scheduler itself, we use list scheduling, the dominant technique; it is relatively straightforward, and produces good code (if the right heuristics are used).

List schedulers can work by producing the instructions from the first one (in the resulting sequence) to the last one, or the other way round. We use a list scheduler that starts with the last instruction, because this works better for our data structures and fits better in the organization of the passes (see Section 4.6).

List scheduling works by selecting one of the leaders (in our case, instructions without successors in the dependence graph) and removes it from the graph. This step is repeated until the graph is empty. The order in which the instructions are removed is the reverse instruction order of the basic block.

One problem in writing a fast instruction scheduler is determining the leaders quickly at every step. We use the following method: When the ML graph is built, the number of dependence references to the node is stored in `ml-count`. When an instruction is selected during scheduling, the reference counts of all the ML nodes on which it depends are decremented; if a reference count reaches 0, the corresponding ML node is added to the set of leader instructions.

Some ML nodes, e.g., nodes that correspond to `I_REG` IL nodes, do not generate any instructions. These nodes get exaggerated reference counts (e.g., in Fig. 6 the have count 2, but only one reference each), so they never become leaders and are never scheduled. This avoids a good amount of work during scheduling and code emission; it also makes dealing with the load delay slots of the MIPS architecture a lot easier.

Our data structure allows finding quickly the ML nodes on which a given ML node depends, but finding the ML nodes that depend on a given ML node is slow. Therefore, we chose a backwards scheduling strategy.

4.3.3 Heuristic

The selection heuristic determines the quality of the schedule. An overview can be found in [SKAH91]. The heuristic we use is:

largest latest execution time (LET) The LET of an instruction is the last cycle when the instruction can start executing without causing any later instruction to stall waiting for the result. Ties are broken by

maximum path length The path length is the sum of the latencies along the longest path from the start of the basic block[3]. This heuristic exposes delay slots early, while there are other instructions to fill them.

The LET is stored in `ml-let`, the path length in `ml-pathlength`. The path length is computed before the scheduler proper starts, whereas the LET is determined during scheduling (therefore the LET values shown in Fig. 6 are not useful). For both computations the latency of the instruction is needed; it is stored in `ml-latency`.

4.4 Local Register Allocation

Local register allocation assigns registers to most ML nodes.

Some nodes already have registers before local register allocation: nodes that compute stack pointers or that compute stack items that reside in registers; and `I_REG` nodes that represent stack pointers or stack items in registers. I.e., all registers that live at the basic block boundaries: `I_REG` nodes for registers that live at the start of the basic block, other nodes for registers that live at its end. Stack items in registers arise from some sort of global or interprocedural register allocation.

These register numbers are put in the `il-reg` field of the IL and are copied to the `ml-reg` field during ML graph construction. This relies on the fact that the IL nodes with predefined registers are roots of trees that are reduced to `reg`. For `I_REG` nodes this is determined by the grammar, for the compute nodes this is ensured, because these nodes are roots of the IL graph.

If an ML node has not yet been assigned a register, its `ml-reg` field contains `regs-unused`.

The local register allocator walks backwards (starting at the instruction that will be executed last) through the array of ML nodes produced by the scheduler. For every ML node, it enters the register that the ML node computes to the list of free registers; some registers, e.g., stack pointers, cannot be freed. Then it takes registers from the free list and assigns them to the yet unassigned input registers of the ML node.

The input registers of an ML node are the `ml-reg` fields of the children (`ml-left` and `ml-right`) of the node. For this reason, we have ML nodes corresponding to the `I_REG` IL nodes, even though these ML nodes do not generate an instruction.

4.4.1 Register Shuffling

One of the problems that we get if we try to keep stack items in registers across basic blocks is: Some registers may live at the start of the basic block and at the end, but don't contain the same thing; at the

[3]The path length is defined differently in [SKAH91].

end a register may contain something that was in a different register at the start.

A particularly nasty version of this problem is this: Assume that x is in `$1` at the start of the basic block and should be in `$2` at the end, whereas y starts in `$2` and ends up in `$1`. The solution for this problem needs three moves. How can we solve the problem in general?

The solution we use is to access the input registers through `I_MOVE` nodes instead of directly through `I_REG` nodes. We insert dependence edges between the move from a register and the node that computes that register; this ensures that the register is copied to a temporary register before it is overwritten.

If an input register has to move to an output register without an intermediate computation, we may have to insert another `I_MOVE` node for moving that register to its final destination, thus allowing the value to move into a temporary if necessary. If we did not add any `I_MOVE` nodes, the additional dependences could form a cycle (and break the scheduler). Adding the minimum number of `I_MOVE` nodes that does not introduce cycles is quite complex and probably slow.

We use the following fast, but suboptimal solution: Consider moving a value from register a to register b, with b containing another value at the start of the basic block. If the destination register of the move from b is unassigned, we set the destination register of the move from a to b; otherwise we introduce another move, with the move from a as source and b as destination. This scheme eliminates the danger of cycles and introduces at most half as many `I_MOVE` nodes as the straightforward approach (always creating an `I_MOVE` node).

Now, we want to avoid actually generating move instructions if possible. This is possible if the move copies the register to itself. So, we enhance the register allocator like this: When it encounters an instruction I whose result is in an input register, the register allocator looks at the ML node for the move from that register. If the move has no result register assigned to it yet, the register allocator assigns the input register (and takes it from the free list). If the move has a result register, this means that there is an instruction behind I that uses the result of the move, so we cannot avoid the move (with this schedule).

There is one more reason for introducing `I_MOVE` nodes: Each operation has only one destination register; if the result of a computation should reside in n registers at the end of the basic block, $n-1$ moves are necessary to get it there.

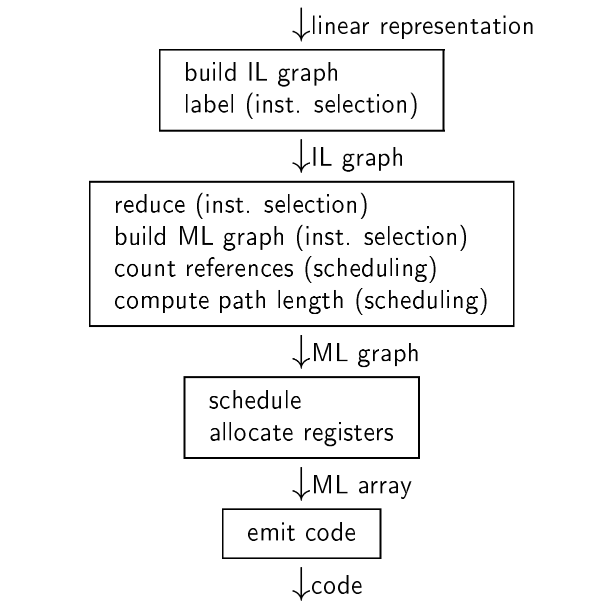

Figure 9: Phases, passes and data structures in our code generator

4.5 Code Emission

Code emission just walks forward through the array of ML nodes, and `EXECUTE`s the XTs stored in the `ml-asm` fields. These words collect the information stored in the `ml-val` and `ml-reg` fields of the node and its children, and call the right assembler word for generating the code.

The code emitter also deals with load delay slots: The code emission action for load instructions checks whether the next ML node depends on the load through a flow dependence; if yes, it generates a `nop` ofter the load. Branch delay slots are handled simply by filling them with `nops`[4].

4.6 Passes

While we would prefer to keep the different phases separate for software engineering reasons, we have integrated them to improve the compile time. This results in the following passes over the data structures (see Fig. 9):

- While building the IL graph, we also perform the labelling part of code selection.
- Then we perform the reducer part of code selection as a recursive graph walk. Upon returning from the recursion, we build the ML graph, count the references to each node and compute the path length for the scheduler.

[4] The obvious approach to utilizing them would be a modified scheduler; however, this approach is not workable, because some ML nodes correspond to more than one instruction (due to other warts in the MIPS architecture), and it would not do to put such a node behind a branch.

relative time		producing threaded code	producing native code
running	threaded code	1.00	4.03
	native code	—	2.95

Figure 10: Relative compile times of different compilers executed with different execution techniques (compiling the RAFTS prototype)

- The next pass performs scheduling and local register allocation, starting at the last instruction, and proceeding to the first.
- Finally, the code is emitted in a forward pass over the scheduled code.

Note that the use of backward scheduling saves two passes.

5 Performance

All timings reported here were taken on a DecStation 5000/200 (25MHz R3000) with 40MB RAM.

5.1 Compilation Speed

The introduction of the faster scheduler and the integrated pass structure has caused a speedup in compilation speed by a factor of two over the results in [EP96], even though we also added the linear intermediate code. You can find a compilaton speed comparison of Gforth's compiler (producing threaded code) and RAFTS in Fig. 10: the current RAFTS prototype (running in native code) compiles itself 2.95 times slower than Gforth (running in threaded code) compiles it. The speedup of the native code version of RAFTS over the threaded code version is 1.37; we attribute this disappointing speedup to the high number of calls to threaded code words.

Figure 11 shows some numbers on the size of our prototype and on the absolute speed. You may wonder about the discrepancy in the number of `COMPILE,`s and linear IR words; note that literals and words with special compilation semantics (e.g., `IF` and `;`) are not COMPILE,d, and they may generate more than one linear IR word (e.g., `ELSE` generates four).

We think that there is still an order-of-magnitude improvement in compile-time possible; however, currently we focus our efforts on interprocedural register allocation.

5.2 Code Quality

The improvements in the code generator had not only postitive effects on the compilation speed, but also on the code quality (better instruction selection and scheduling). We have also implemented our suggestion (in [EP96]) to keep the top of stack (or more stack items) in registers across basic block boundaries, with a fixed mapping (i.e., not a real global or interprocedural register allocation, more like a calling convention).

Figure 12 shows the results, relative to the combination of forth2c [EM95] and gcc-2.5.8 -O2. The number of registers seems to have little influence here, but, based on the results in [Ert95], we think that this is due to the choice of benchmarks (most of which do most work in simple counted loops), and is not valid in general. We think that in general the results look more like the results for the fib benchmark, i.e., a few fixed registers help, but more hurt (because they have to be moved around whenever a basic block changes the stack depth).

Overall, the results are quite encouraging: On one benchmark RAFTS beats the forth2c/gcc, on another it can keep up, and the worst one has a slowdown factor of 1.75. However, these bench-

	items	cycles/item
compile-time (s)	9.89	25000000
lines	6411	38566
`COMPILE,`	6214	39789
linear IR words	15940	15511
IL nodes	32520	7603
ML nodes	24687	10015
native code cells	26984	9162

Figure 11: Size of the compiler and compilation speed (absolute and in cycles/item); this is not a split of the time into various subtasks.

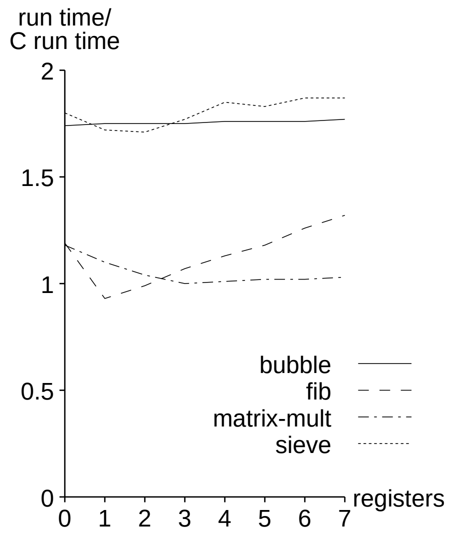

Figure 12: Code quality of RAFTS with varying numbers of stack items in registers, scaled to the run-time of code produced by `gcc-2.5.8 -O2`

marks are very small, and the results for realistic programs may be different. This shows the necessity of larger benchmark programs for evaluating sophistsicated compilers.

6 Conclusion

We have presented some of the more interesting implementation details of our RAFTS prototype:

- How we combine the goals of integration with the threaded code system and efficient word-specific compilation actions,
- A stack-based threaded code intermediate representation for programs that allows delaying code generation until `EXECUTE`.
- The data structures and algorithms that we use in basic block code generation.

Acknowledgements

Bernd Paysan, Manfred Brockhaus and Andreas Krall provided helpful comments on earlier versions of this paper.

References

[DS84] L. Peter Deutsch and Allen M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Principles of Programming Languages (POPL'84)*, pages 297–302, 1984.

[EM95] M. Anton Ertl and Martin Maierhofer. Translating Forth to efficient C. In *EuroForth '95 Conference Proceedings*, Schloß Dagstuhl, Germany, 1995.

[EP96] M. Anton Ertl and Christian Pirker. RAFTS for basic blocks: A progress report on Forth native code compilation. In *EuroForth '96 Conference Proceedings*, St. Petersburg, Russia, 1996.

[Ert92] M. Anton Ertl. A new approach to Forth native code generation. In *EuroForth '92*, pages 73–78, Southampton, England, 1992. MicroProcessor Engineering.

[Ert95] M. Anton Ertl. Stack caching for interpreters. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 315–327, 1995.

[FHP92] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG — fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, April 1992. Available from ftp://kaese.cs.wisc.edu/pub/burg.shar.Z.

[Pro95] Todd A. Proebsting. Burs automata generation. *ACM Transactions on Programming Languages and Systems*, 17(3):461–486, May 1995.

[SKAH91] Mark Smotherman, Sanjay Krishnamurthy, P. S. Aravind, and David Hunnicutt. Efficient DAG construction and heuristic calculation for instruction scheduling. In *MICRO-24, 24^{th} Annual Intl. Symp. on Microarchitecture*, pages 93–102, 1991.