Automatic Generation of Executable Data Structures

Nobuhisa Fujinami

Sony Computer Science Laboratories, Inc.

1 Introduction

Run-time code generation improves programs by generating machine code specific to values that are unknown until run time. Such values are called *run-time constants*. Examples are intermediate computation results and user inputs. Executing the generated code enough times will recoup the cost of run-time code generation.

Massalin's executable data structures (EDSs) method used in Synthesis Kernel [1] [2] is a kind of run-time code generation. It reduces the traversal time of data structures by implementing the traversal operation as self-traversal code stored in each data structure. The code is specialized to both the traversal operation and the containing data structure. It is attractive but not easy to use, because it is implemented using hand-written templates in assembly language. Only well-trained programmers can use it.

This paper shows that EDSs can be viewed as an optimized implementation of a group of objects and proposes a method to generate them automatically from the source program in an object-oriented language. It assumes that the group consists of one representative object (container) and other objects (element). The optimized program generates code for methods of the elements at run time. The code is optimized for both the element values and the caller method of the container.

The author has already proposed a system to automate run-time code generation in object-oriented languages [3] [4]. The method proposed in this paper is an extension of the proposal. It is currently implemented as a preprocessor of C++ compilers. It focuses on instance variables whose values are invariant or quasi-invariant. For methods that use such values, it generates specialized code generators in C++[1]. It also inserts code fragments to invoke them and to invoke or invalidate the generated code. Whether the embedded instance variables are invariant and changes to the variables are automatically detected using the encapsulation mechanism of the object-oriented language[2]. Since run-time code generators are specialized for particular methods, code generation is light-weight. And since each instance has its own generated code for its methods, there can be multiple optimized versions for one method. Memory management for generated code is simple, because it can be left to the object construction/destruction mechanism of object-oriented languages.

---

[1] When invoked, the code generators write machine instructions directly into memory.

[2] If the embedded values are changed, the code is invalidated and then regenerated.

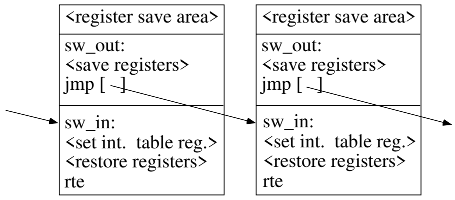

Fig. 1. Task queue

The remainder of this paper is organized as follows: Section 2 overviews the idea of the EDSs method. Section 3 illustrates the optimization of a group of objects, and Section 4 describes a general method. Section 5 presents a preliminary evaluation of the method. Section 6 compares this paper with related work. Finally Section 7 concludes the paper.

2 Executable Data Structures (EDSs)

The EDSs method reduces the traversal time of data structures. It is useful for data structures that are frequently traversed in a preferred way. If the data structures are nodes linked by pointers, the method stores node-specific traversal code along with the data in each node and makes the data structure *self-traversing.*

Figure 1 is an example of a task queue managed by a simple round-robin scheduler. Each element in the queue contains two short sequences of code: `sw_out` and `sw_in`. The `sw_out` saves the registers and branches to the next task's `sw_in` routine (in the next element in queue). The `sw_in` restores the new task's registers, installs the address of its own `sw_out` in the timer interrupt vector table, and resumes processing.

The `sw_out` and `sw_in` routines are optimized to the particular task and move only needed data. For example, if floating point operations have not been used in the task, floating point context is not saved/restored.

When adding a new element (task) to the queue, `sw_out` and `sw_in` routines specialized to the new element are generated using templates in assembly language, and the jump address in the `sw_out` routine in the previous element is set.

3 Optimizing a group of objects

In the rest of this paper, C++ notation is used for object-oriented terminology. For example, methods are called member functions, and instance variables are called data members. Assume also that all source code for implementing the object group is available.

```
class String {
 private:
  char s[MAXLEN];
  int len;
 public:
  String(char const *p);
  int match(char const *p);
};

class StringLink {
 public:
  StringLink *next;
  String s;
  StringLink(char const *p): s(p) {}
  int match(char const *p) { return s.match(p); }
};

class StringList {
 private:
  StringLink *top;
 public:
  StringList(): top(NULL) {}
  void add(char const *p);
  int search(char const *p);
 private:
  int search(char const *p,StringLink *q);
};
```

Fig. 2. Class definitions for linked lists

The target of the EDSs method is a group of data structures frequently traversed in the similar way. If the group is implemented in a object-oriented language, the usual practice is to make one container object for the group. All operations on the group are invoked through the container. Various kinds of data structures, such as linked lists, binary trees, or hash tables can be implemented as this type of object groups.

If the conventional framework of run-time optimization is applied to the whole data structure represented by the objects, all elements and their relationships are regarded to be run-time constants. The whole generated code becomes invalid each time when the relationships of the elements changes. Since the optimization result is a set of EDSs in the proposed method, the generated code allows partial modification according to the changes of relationships of the element objects.

For instance, Figure 2 shows class definitions for a linked list in C++. Each element holds a string object. First, the list is empty. Member function `add` puts an element at the top of the list. Member function `search` tests if the parameter string is in the list or not.

Since class `StringList` does not import any elements, the container `StringList` can only call member function `match` of elements `StringLink` instantiated within the container. Member function `match` can be adapted to `search` (see Figure 3 for its definition). Member function `search` can be implemented as distributed

```
int StringList::search(char const *p) { return search(p,top); }
int StringList::search(char const *p,StringLink *q) {
  if(q==NULL) return 0;
  if(q->match(p)) return 1;
  return search(p,q->next);
}
```

Fig. 3. Member function search

```
; ---code in container---
StringList_search:
        <function prologue>
        MOV ECX,this
        MOV ESI,[ECX].top             ; ESI <- address of first string
        CALL DWORD PTR [ECX].top_code ; call machine code for first string
        <function epilogue>
; ---code in element STRING1---
STRING1:<compare with first string and jump to OK1 or NG1>
OK1:    MOV EAX,1
        RET                           ; return 1
NG1:    MOV EAX,[ESI].next_code
        MOV ESI,[ESI].next            ; ESI <- address of next string
        JMP EAX                       ; jump to code for next string
; ---code in element STRING2---
STRING2:<compare with second string and jump to OK2 or NG2>
        <similar to STRING1 (omitted)>
        :
        :
; ---termination code---
NOSTRING:XOR EAX,EAX
         RET                          ; return 0
```

Fig. 4. Optimized member function `search`

(Intel x86 assembly language notation is used for readability. Actual code is written in memory as a binary image.)

code fragments in elements as shown in Figure 4. The tail recursion of `search` is completely unrolled and the transition to the next element operation is implemented as a jump instruction to the code for the next element. The destination of the jump instruction is updated when the list is reconfigured.

Each of code fragments `STRING1`, `STRING2`, $\cdots$ is specialized to both its corresponding string and member function `search`. The code in `<compare with ...>` for `STRING`n is the result of specializing member function `match` with respect to the n-th string in the list. The rest is a specialized form of `search`.

Data members `top_code` and `next_code` in the code in Figure 4 are newly added members to classes `StringList` and `StringLink`. They represent addresses of the code for elements. Member functions that modify the linked list should update these members. When an instance of class `StringList` is initialized, the address of label `NOSTRING` in Figure 4 should be assigned to member `top_code`. These assignments are parallel to the corresponding members `top` and `next`. Figure 5 shows member function `add` before and after the optimization.

```
// Before optimization
void StringList::add(char const *p) {
  StringLink *l=new StringLink(p);
  l->next=top; top=l;
}

// after optimization
void StringList::add(char const *p) {
  StringLink *l=new StringLink(p);
  <generate code and set its address to qqcode>
  l->next=top; top=l;
  l->next_code=top_code; top_code=qqcode;
}
```

Fig. 5. Member function `add`

4 Automatic optimization

This section describes a method of automating the optimization illustrated in the previous section. It decides the applicability of the optimization, generates run-time code generators, and modifies the program to use the code generators and the generated code. Both the container and elements are assumed to have invariant data members.

4.1 Deciding applicability

To use the EDSs method, some member functions of the container should repeatedly use its elements. This paper focuses on tail recursions as the iteration constructs and on traversing links as a way to find next elements in iteration. Non-tail recursions, such as full tree traversals, are not considered because they require backward links of the generated code[3]. Loops are not considered either. In most cases, loops can be rewritten into tail recursions by analyzing loop control variables.

The condition for finding iteration by tail recursions is as follows:

- There exists a container's member function, say r, that calls itself as tail recursions (there can be multiple calling sites).
- One of the formal parameters, say p, of r is a pointer to an element[4].
- Function r uses invariant data members of the object pointed to by p or invokes member functions[5] that use such data members.
- There exists a container's member function, say s, that calls r.

[3] Traversing all the tree nodes must be rare cases, because tree structures are useful in reducing the number of nodes to be visited.

[4] Since reference types can be rewritten using pointer types, they are treated in the similar way.

[5] The function body will be inlined.

It is out of the scope of this optimization to invoke r directly from outside. Container classes normally have member functions equivalent to s to encapsulate internal implementation.

Though it is possible to deal with imported elements if the timing of importation is known, they are not considered for simplicity. All the elements used in s and r should be instantiated in the member functions of the container. One sufficient condition is to confirm all of the following:

- Let *descendants* of an element be elements that can be reached by 0-or-more-steps traversal of the data members whose type is a pointer to an element.
- Let *element variables* be data members of container or elements whose type is a pointer to an element.
- The actual parameter for p to call r from s should be a descendant of an element pointed to by a `private` data member of the container.
- The actual parameter for p to call r from r should be a descendant of either an element pointed by caller's p or a `private` data member of the container.
- If the container has a member function that modifies element variables, the new values should be either null pointers, pointers to descendants of objects pointed to by the container's element variables, or pointers to elements instantiated in the member function. If the member function is r, the new values may be pointers to descendants of an object pointed to by p.
- Elements should not have a member function that modifies element variables.
- The container should not export non-`const` pointers[6] to elements. That is, it should not have a member function to return them or to write them into globally accessible variables or memory addresses specified by arguments.
- Elements should not have a member function to write non-`const` pointers to elements into globally accessible variables.

Program transformation by dataflow analysis, of course, allows cases where the actual parameter is a temporary variable that has a value of an element variable. Thus, the above conditions are general enough, not particularly hard to satisfy.

4.2 Generating code generators

This subsection describes a way to generate run-time code generators specialized to s and r. First, a code generator for s with first level inlining of r is generated (possible with an existing method) (Figure 6(a)). Invariant data members of the container and the object pointed to by p are treated as run-time constants in the code generator. Here, optimizations across the s-r boundary are restricted. For example, constant propagation, code motion, and common subexpression elimination across the boundary are disabled[7]. This restriction is for reconfiguration of the data structure (code for r is replaced).

[6] To cast a pointer to `const` into a pointer to non-`const` is considered to be illegal, because it is not safe to modify an object through such a pointer[5].

[7] Constant propagation from s to r is allowed.

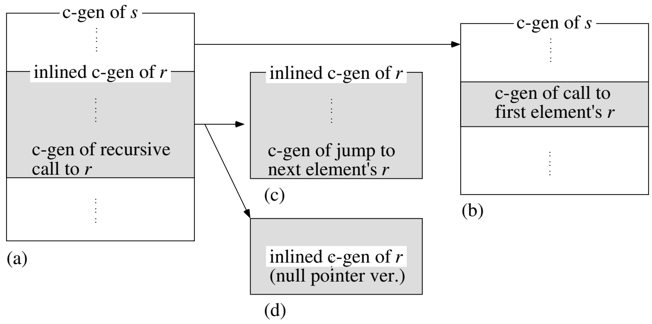

Fig. 6. Construction of code generators

("c-gen" is short for "code generator".)

The resulting code generator is separated into that for s and r. The hole in s (after r's part is extracted) is filled with a code generator to invoke the code for the element pointed to by the actual parameter (Figure 6(b)). An *address variable* described in the next subsection gives the code address.

Since r normally checks whether or not p is a null pointer, the code generator for r is duplicated: one optimized assuming p is a null pointer (generating code such as `STRING1` in Figure 4) and one optimized assuming p is not a null pointer (generating code such as `NOSTRING` in Figure 4). The tail recursion part of r is replaced with a code generator to jump to the code for the element pointed to by the actual parameter (Figure 6(c),(d)).

4.3 Using code generators and generated code

For each element variable, a new data member called an *address variable* is added to represent the address of the code for the element pointed to by the element variable. Member functions of the container are modified to assign the addresses of the code to address variables when the corresponding element variable is assigned.

The member function s is replaced with the following procedure:

- Check if the code for s is already generated. If not, invoke the code generator for s.
- Invoke the generated code for s.

Invocation of the non-null version of the code generator for r is embedded where the element is instantiated. Invariant data members of the instantiated object are treated as run-time constants. After the invocation, the address of the generated code is assigned to the address variable corresponding to the element variable to which the pointer to the instantiated object is assigned.

Since the null-pointer version of the code generator for r cannot use data members of an object pointed to by p, the resulting code must be independent

program	exec. time (sec) / ratio	code gen. cost (msec)	code size (bytes)
P1. original	8.92 / 1.00	-	-
P2. conventional	8.76 / 0.98	2.69	1348
P3. manual inlining	3.33 / 0.37	-	-
P4. proposed	2.35 / 0.26	2.72	1251

Table 1. Linked list search results

of elements. Thus invocation of the code generator is embedded into the constructor of the container. An assignment of the address of the generated code to an address variable is embedded where a null pointer is assigned to the corresponding element variable.

4.4 Deleting generated code

There are two types of code deletion: Deletion of code pointed to by an address variable is performed where the element pointed to by the corresponding element variable is deleted. The code of the null-pointer version of r generated in the constructor of the container is deleted in its destructor.

5 Evaluation

Four programs are compared to evaluate the method proposed in this paper. All are implementations of a linked list of strings described in Section 3. P1 is compiled normally. P2 is processed by a preprocessor [3] using an element-wise run-time optimization. P3 is the result of manual elimination of tail recursion and manual inlining of member function `match`. P4 is the result of the method proposed in this paper.

The target machine is an NEC PC-9821La10/8 (Pentium 100MHz, 23MBytes of memory, no second cache). The operating system is Windows 95. The compiler is BCC32A.EXE (a command line compiler of Borland C++ 4.5J). Compiler options are "-5 -O2 -vi" (optimized for Pentium, optimized for speed, inlining enabled).

Table 1 lists the execution times to search a list of ten elements for both existing and non existing strings a million times. Each value is an average of ten executions.

Since the conventional method can optimize only member function `match` to its string and cannot eliminate function call overhead, speedup of P2 is very small. Manual inlining with tail recursion elimination (P3) works well, but since the inlined member function `match` is not optimized to its string, there is room for improvement. The proposed method allows all of these optimizations, and P4 shows the best performance.

Table 1 also lists the run-time code generation costs and generated code sizes for P2 and P4. Several hundred times of iteration recoups the cost in P4.

6 Related Work

A number of research papers report methods of making it easy and practical to use run-time code generation. Among the methods are description languages for run-time code generators [6], a run-time partial evaluation system for binary code [7], and automatic generation system of run-time code generators [8] [9]. But they deal with single functions, parts of functions, or program portions that can be arranged into single functions (e.g. using inlining).

One research paper mentions automatic generation of EDSs. Subsection 4.4 of [10] suggests that applying run-time partial evaluation can perform the same optimization as using EDSs. An example used in the subsection is a scheduler expressed in pseudo-code:

```
task *crt_tsk;

sched() {
  save_regs(crt_tsk);
  crt_tsk=crt_tsk->next;
  load_regs(crt_tsk);
  jmp_to(crt_tsk->PC);
}
```

Through specialization with respect to a given circular task queue, the resulting program is

```
sched_1() {
  save(FP1); save(FP4); // save registers for current task
  load(FP2); load(FP3); // load registers for following task
  jmp(0x....);
}
```

The second command in `sched`, which dereferences the pointer to the next task to be activated, is eliminated. Other commands are specialized with respect to the current and following tasks.

This result, however, though the example is overly simplified, is very far from the concept of EDSs. It does not make the data structure *self-traversing*. It fixes the order of the tasks. If a task's state changes from active to inactive, the context switch routines for both the previous and the following task should be changed. This is because the specialization views the whole queue as a run-time constant, like the former (trivial) type of optimizations described in Section 3. It disables the flexibility of linked structures. In the EDSs method, flexibility is preserved, and changing a task's state requires only the jump addresses in the switch code to be updated.

There may be some techniques, such as indirect jumps, that make the method in [10] flexible enough. But the programmer must carefully rewrite routines for updating the task switch routine. So it is not easy to use.

Since the method in this paper generates EDSs, the operation to update the links is light-weight. And this updating operation is automatically derived from the original source program.

7 Conclusion

This paper proposed a method of generating EDSs automatically from the source code in object-oriented languages, viewing them as an optimized implementation of a group of objects. With the help of an encapsulation mechanism of object-oriented languages, analyzing only the source code to implement the object group proves the method applicability. Generating code generators for EDSs is an extension of the conventional technique of generating run-time code generators. The preliminary evaluation shows that the proposed method is better than previous methods or the manual source level optimization.

The author will extend the method to deal with quasi-invariant data members. He will also implement a reverse operation of tail recursion elimination to accept iteration by loops.

References

1. Calton Pu, Henry Massalin, and John Ioannidis. The Synthesis kernel. *Computing Systems*, Vol. 1, No. 1, pp. 11–32, Winter 1988.
2. Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Graduate School of Arts and Sciences, Columbia University, April 1992.
3. Nobuhisa Fujinami. Automatic Run-Time Code Generation in C++. In Yutaka Ishikawa, Rodney R. Oldehoeft, John V.W. Reynders, and Marydell Tholburn, editors, *LNCS 1343: Scientific Computing in Object-Oriented Parallel Environments. First International Conference, ISCOPE 97 Proceedings*, pp. 9–16, December 1997.
4. Nobuhisa Fujinami. Determination of Dynamic Method Dispatches Using Run-time Code Generation. In Xavier Leroy and Atsushi Ohori, editors, *LNCS 1472: Types in Compilation. Second International Workshop, TIC'98 Proceedings*, pp. 253–271, March 1998.
5. Bjarne Stroustrup, editor. *The C++ Programming Language, 2nd edition*. Addison Wesley, 1991.
6. Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C: A language For High-Level, Efficient, and Machine-independent Dynamic Code Generation. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 258–270, January 1996.
7. Curtis Yarvin and Adam Sah. QuaC: Binary Optimization for Fast Runtime Code Generation in C. Technical Report UCB//CSD-94-792, University of California Berkeley, Department of Computer Science, 1994.
8. Mark Leone and Peter Lee. A Declarative Approach to Run-Time Code Generation. In *Workshop Record of WCSSS'96: The Inaugural Workshop on Compiler Support for System Software*, pp. 8–17, February 1996.
9. Brian Grant, Markus Mock, Matthai Philipose, Graig Chambers, and Susan J. Eggers. DyC: An Expressive Annotation-Directed Dynamic Compiler for C. Technical Report 97-03-03, Department of Computer Science and Engineering, University of Washington, 1997.
10. Charles Consel, Luke Hornof, François Nöel, and Nicolae Volanshi. A Uniform Approach for Compile-time and Run-time Specialization. Technical Report No. 2775, INRIA, January 1996.