Efficient Incremental Run-Time Specialization for Free

Renaud Marlet Charles Consel Philippe Boinot

IRISA / INRIA - Universit de Rennes 1
Compose project
Campus universitaire de Beaulieu
35042 Rennes cedex, France
{marlet,consel,pboinot}@irisa.fr

Abstract

Availability of data in a program determines computation stages. Incremental partial evaluation exploit these stages for optimization: it allows further specialization to be performed as data become available at later stages. The fundamental advantage of incremental specialization is to factorize the specialization process. As a result, specializing a program at a given stage costs considerably less than specializing it once all the data are available.

We present a realistic and flexible approach to achieve efficient incremental run-time specialization. Rather than developing specific techniques, as previously proposed, we are able to re-use existing technology by iterating a specialization process. Moreover, in doing so, we do not lose any specialization opportunities. This approach makes it possible to exploit nested quasi-invariants and to speed up the run-time specialization process.

Our approach has been implemented in Tempo, a specializer for C programs that is publicly available. A preliminary experiment confirms that incremental specialization can greatly speed up the specialization process.

1 Introduction

Different stages of computation can be identified in a program, depending on the availability of data. Code corresponding to later stages can often be optimized by performing in advance the computations depending on data available at earlier stages, i.e., by factorizing some computations from late stages into earlier stages.

To appear in the proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation, May 1999, Atlanta, Georgia.

1.1 Staging in Partial Evaluation

Staging is the essence of partial evaluation: it traditionally makes explicit two stages (binding times), namely static (early stage) and dynamic (late stage).

Consider[1] a program p which normally computes a result out in one stage from input $\texttt{in}_s$ and $\texttt{in}_d$.

$$\texttt{out} \;=\; [\![\texttt{p}]\!]\ \texttt{in}_s\ \texttt{in}_d$$

Given a binding-time division of p's input, a program specializer spec splits computations into two stages:

$$\begin{aligned}\texttt{p-res} &= [\![\texttt{spec}]\!]\ \texttt{p}\ \text{`}\texttt{SD}\text{'}\ \texttt{in}_s\\ \texttt{out} &= [\![\texttt{p-res}]\!]\ \texttt{in}_d\end{aligned}$$

In the first stage, when building the residual (i.e., specialized) program p-res, computations depending only on input data $\texttt{in}_s$ are performed. Remaining computations needed to obtain out are performed in a second stage.

1.2 Staging in Loop Nests

Staging also arises in nested loops. The deeper is the nesting, the later is the stage (and the more frequent the execution). Consider for example the program below.

```
for i = ...             (stage 1)
  for j = ...           (stage 2)
    for k = ...         (stage 3)
      f(i,j,k)          (stage 3)
    end                 (stage 3)
  end                   (stage 2)
end                     (stage 1)
```

Variable i does not vary inside the body of the first, outmost loop. It is called a *quasi-invariant*. Any computation in f depending only on i can be performed before the second, middle loop is executed, thus avoiding being recomputed at each iteration of variables j and k. Similarly, any computation depending only on

[1] Notations are borrowed or adapted from [12, 17]. In addition, we note '$\texttt{S}^i\texttt{D}^j$' a binding-time sequence consisting of i occurrences of S (static) and j occurrences of D (dynamic).

i and j can be performed before the third, innermost loop is executed. Informally, this factorization could be obtained with the following specializations.

```
for i = ...
  f_i = spec(f,i)
  for j = ...
    f_i_j = spec(f_i,j)
    for k = ...
      f_i_j(k)
    end
  end
end
```

This corresponds to an *incremental specialization* process: further specialization are performed as data for later stages become available.

1.3 Benefits of Incremental Specialization

Just as ordinary specialization, incremental specialization is beneficial only if specialized functions are used enough times to amortize the time to construct them.

Specialization can be performed at compile time as a source-to-source transformation. For example, a generic microprocessor simulator can be first specialized with respect to a given instruction set, yielding a simulator dedicated to a processor, and then be further specialized with respect to a program to speed up its simulation time. Similarly, a generic program-analysis engine (e.g., Z1 [36], BANE [2]) could be specialized with respect to a language (abstract interpreter or equation generator) and an analysis (precision and abstract domains). Other applications include meta-interpreters taking as successive input a language definition, a program and its data, and the generation of a compiler generator capable of supercompilation from a two-level driving interpreter [12, 13].

However, compile-time specialization, whether or not incremental, is limited. For example, it cannot be applied to the above loop nest because of the compilation overhead at each stage would make it impractical. To eliminate this overhead, we must resort to run-time specialization, i.e., run-time code generation. In the loop nest example, the benefit of incremental specialization varies according to the amount of computations depending on the quasi-invariants (i.e., the execution cost of pre-computed static expressions), the number of loop iterations (i.e., the amount of re-use) and the time taken for code generation. Besides loop nests, incremental run-time specialization can be used to speed up staged computations that are inherently dynamic such as configuring a system with respect to run-time parameters.

In fact, the Synthetix project has been advocating the use of incremental specialization to optimize operating systems [8]. They used the technique successfully to optimize the HP-UX file system by exploiting staged invariants of this subsystem [29].

Also, a combination of compile-time and run-time specialization was used by Volanschi to optimize the Chorus IPC [34].

1.4 This paper

In this paper, we study how a practical multi-level specialization process can be derived from a simple two-level model. In particular we show how Tempo, a two-level run-time specializer for C, can be turned into an efficient incremental partial evaluator. Our contributions are the following:

- We present a realistic and flexible approach to achieve efficient incremental run-time specialization.
- This provides some practical insight into the nature of incremental partial evaluation.
- Our approach is simpler than existing techniques and relies on well-known and available technology.
- It is implemented in an existing program specializer named Tempo and a preliminary experiment confirm that it can considerably speed up the specialization process.

The paper is organized as follows. Section 2 presents two approaches for achieving incremental partial evaluation, namely "native" multi-level specialization and iterated two-level specialization. Section 3 describes our implementation of the latter approach in Tempo. Section 4 analyzes our iterative specialization approach and Section 5 compares it with related work. Section 6 lists ongoing and future work. Section 7 concludes.

2 Incremental Partial Evaluation

Most offline partial evaluators rely on the concept of generating extension [16]. A *generating extension* p-gen is a generator of specialized p programs.

$$\begin{aligned}\texttt{p-res} &= [\![\texttt{p-gen}]\!]\ \texttt{in}_s \\ \texttt{out} &= [\![\texttt{p-res}]\!]\ \texttt{in}_d\end{aligned}$$

A generating extension is generally obtained using a *compiler generator* cogen given a binding-time specification.

$$\texttt{p-gen} = [\![\texttt{cogen}]\!]\ \texttt{p 'SD'}$$

A compiler generator provides a specializer:

$$[\![\texttt{spec}]\!]\ \texttt{p 'SD' in}_s \stackrel{\text{def}}{=} [\![[\![\texttt{cogen}]\!]\ \texttt{p 'SD'}]\!]\ \texttt{in}_s$$

The question is: how can this two-level staging of computations be extended to n levels in the case of a program p with n arguments?

$$\texttt{out} = [\![\texttt{p}]\!]\ \texttt{in}_1 \cdots \texttt{in}_n$$

(Without loss of generality, we assume that `p` has exactly n inputs, provided in the staging order $\mathtt{in}_1, \ldots, \mathtt{in}_n$.)

There exists basically two main approaches to achieve incremental partial evaluation [13]. These approaches extend the concept of generating extensions to multi-level specialization.

2.1 Multi-Level Generating Extension

The idea of multi-level offline specialization [11, 13] can be described as follows.

> An n-level generating extension is a program that produces $(n-1)$-level generating extensions.

The base case is a two-level generating extension, which corresponds to the classic definition of `p-gen` given above.

An n-level generating extension $\mathtt{p\text{-}mgen}_n$ for program `p` is used as follows.

$$\begin{array}{rcl} \mathtt{p\text{-}mgen}_{n-1} & = & [\![\mathtt{p\text{-}mgen}_n]\!]\ \mathtt{in}_1 \\ & \vdots & \\ \mathtt{p\text{-}mgen}_1 & = & [\![\mathtt{p\text{-}mgen}_2]\!]\ \mathtt{in}_{n-1} \\ \mathtt{out} & = & [\![\mathtt{p\text{-}mgen}_1]\!]\ \mathtt{in}_n \end{array}$$

At step i, when computing the $(n-i)$-level generating extension $\mathtt{p\text{-}mgen}_{n-i}$, only computations of `p` that depend on stages i and before (i.e., on arguments $\mathtt{in}_1, \ldots, \mathtt{in}_i$) are performed.

As an example, consider the loop nest pattern presented in the introduction. A three-level generating extension `f_mgen3` for function `f` can be used as follows.

```
for i = ...
  f_mgen2 = f_mgen3(i)
  for j = ...
    f_mgen1 = f_mgen2(j)
    for k = ...
      f_mgen1(k)
    end
  end
end
```

The fact that some computations at a given stage do not vary in deeper stages in now made explicit.

2.2 Multi-Level Compiler Generator

To construct multi-level generating extensions, Glck and Jrgensen [12] propose to extend the two-point binding-time domain $\{\mathtt{S}, \mathtt{D}\}$ to a domain $\{1, \ldots, n\}$ of cardinality n. A multi-level binding time i in this domain corresponds to input that is available only at stage i and subsequently. Two-level binding-time analyses (BTA) can be extended to multi-level binding-time analyses (MBTA) in order to treat n stages. For example, "$\sqcup$" in the "$\mathtt{S} \sqsubset \mathtt{D}$" lattice becomes "max" in the "$1 \leqslant \cdots \leqslant n$" lattice.

A multi-level compiler generator `mcogen` can then be defined to exploit this multi-level binding-time analysis and produce multi-level generating extensions.

$$\mathtt{p\text{-}mgen}_n \quad = \quad [\![\mathtt{mcogen}]\!]\ \mathtt{p}\ `\mathtt{1}\cdots n\text{'}$$

However, defining an `mcogen` is not a trivial task [13].

2.3 Iterated Two-Level Specialization

Instead of specifically developing a multi-level binding-time analysis and a multi-level compiler generator `mcogen`, we consider an alternative construction called *incremental self-application* [11, 12] or *incremental generation* [13].

To first get some intuition, consider again the loop nest case, given only a two-level generating extension.

```
for i = ...
  for j = ...
    f_res = f_gen(i,j)
    for k = ...
      f_res(k)
    end
  end
end
```

To further factorize computations depending only on `i` inside the `j` loop, the function `f_gen` itself could be specialized.

```
for i = ...
  f_gen_res = f_gen_gen(i)
  for j = ...
    f_res = f_gen_res(j)
    for k = ...
      f_res(k)
    end
  end
end
```

Specialization is advantageous only if the residual code is used enough times to compensate for the building cost. In our example, whether to further specialize `f_gen` or not depends on the number of iterations on `j`, the amount of computations depending only on `i` in `f_gen`, and the time required for generating `f_gen_res`.

This re-specialization further stages computations that were previously declared as static. It suggests an alternative to n-level generating extensions:

$$\mathtt{p\text{-}igen}_n \quad = \quad (\mathtt{p\text{-}igen}_{n-1})\text{-}\mathtt{gen}$$

More precisely, given a two-level `cogen` and a program `p`, we define:

$$\begin{array}{rcl} \mathtt{p\text{-}igen}_1 & = & \mathtt{p} \\ \mathtt{p\text{-}igen}_2 & = & [\![\mathtt{cogen}]\!]\ \mathtt{p\text{-}igen}_1\ `\overbrace{\mathtt{S}\cdots\mathtt{S}}^{n-1}\mathtt{D}\text{'} \\ & \vdots & \\ \mathtt{p\text{-}igen}_n & = & [\![\mathtt{cogen}]\!]\ \mathtt{p\text{-}igen}_{n-1}\ `\mathtt{SD}\text{'} \end{array}$$

Program $\texttt{p-igen}_n$ achieves multi-level specialization in the sense that:

$$\texttt{p-igen}_n \text{ is an } n\text{-level generating extension}$$

This statement says that iterated applications of $\texttt{p-igen}_n$ to the sequence of inputs $\texttt{in}_1, \ldots, \texttt{in}_n$ yields the result `out`. The proof is by induction on the level n; the induction hypothesis is applied to program $\texttt{p-igen}_2$.

Another way to understand $\texttt{p-igen}_n$ is the following. Let `spec` be the specializer defined from `cogen`. Then, for all $0 \leqslant i < n$,

$$[\![\cdots[\![\texttt{p-igen}_n]\!]\,\texttt{in}_1\cdots]\!]\,\texttt{in}_i = [\![\texttt{spec}]\!]\ \texttt{p-igen}_{n-i}\ \text{`}\texttt{S}^i\texttt{D}\text{'}\ \texttt{in}_1\cdots\texttt{in}_i$$

The proof is by induction on i.

These facts are important to capture the nature of this iterated generating extension: each additional level in the construction merely consists in specializing the preceding level. In particular, compared to traditional two-level specialization, incremental specialization does not discover more specialization. Indeed, consider the case where $i = n-1$ and compare it with the two-level specialization of `p` with binding times $\text{`}\texttt{S}^{n-1}\texttt{D}\text{'}$:

$$[\![\cdots[\![\texttt{p-igen}_n]\!]\,\texttt{in}_1\cdots]\!]\,\texttt{in}_{n-1} = [\![\texttt{spec}]\!]\ \texttt{p}\ \text{`}\texttt{S}^{n-1}\texttt{D}\text{'}\ \texttt{in}_1\cdots\texttt{in}_{n-1}$$

Thus, given a two-level specialization, further staging the static inputs does not alter final the specialized program. What incremental specialization does is just to optimize the specialization process itself. Incrementality may thus lower the specialization break-even point, which is the number of times that the specialized program must be executed to amortize the cost of specialization. This is especially useful at run time (as opposed to compile time) because specialization must be as fast as possible.

3 Implementation

This iterated approach for incremental specialization has been implemented using Tempo.

Tempo is an offline partial evaluator for C programs [5, 6]. It allows programs to be specialized both at compile time and run time. It has been applied in various domains, including operating system and networking [22], domain-specific languages [31, 32], software architectures [20], and numerical computation [18]. Tempo is publicly available[2].

In this section, we describe how Tempo can be applied multiple times to the run-time generating extensions that it generates, thus yielding incremental run-time specialization. The basic concepts and implementation of the run-time specializer have been described elsewhere [7, 23, 24]. Still, we mention here specific features of Tempo that make incremental specialization particularly fast.

[2] Tempo's home page: `http://www.irisa.fr/compose/tempo`

```
int dotprod(int size, int u[], int v[])
{
  int i, res = 0;
  for(i = 0; i < size; i++)
  {
    res += u[i] * v[i];
  }
  return res;
}
```

Figure 1: Inner product function `dotprod`

```
int dotprod(int size, int u[], int v[])
{
  int i, res = 0;
  for(i = 0; i < size; i++)
  {
    res += u[i] * v[i];
  }
  return res;
}
```

Figure 2: [\![`bta`]\!] `dotprod` ‘`SSD`’

We explain the process of incremental specialization only through an example: the function `dotprod` given in Figure 1.

3.1 Binding-Time Analysis

Assume arguments `size` and `u[]` of function `dotprod` are static. The result of the corresponding binding-time analysis is shown in Figure 2. Dynamic expressions and statements are underlined; the other constructs are assumed static. (Tempo displays similarly the results of its BTA, using colors for different binding times.)

A compile-time generating extension based on this binding-time division would[3] look like the function Figure 3. The static slice of function `dotprod` determines the control of the specialization process; the dynamic slice is printed into some specialization output stream. Dynamic code may contain *holes* (denoted with `%d`) to be filled by the result of static expressions; these fragments are called *templates*.

Note that, even though `0` is a static constant, the initial definition of `res` (i.e., the assignment "`res = 0;`") has to be residualized because the variable becomes dynamic in the body of the loop [15]. In fact, rather than assigning the binding time "`res = 0;`" as one would expect since `0` is static, Tempo treats it a fully dynamic statement: "`res = 0;`". The reason is that it is useless to consider literal constants as static when they are in a dynamic context: they cannot be exploited for any

[3] Tempo's compile-time specializer does not rely on a generating extension technology; it interprets specialization actions.

```
dotprod_ctgen(int size, int u[])
{
  int i;
  printf("int dotprod_res(int v[])");
  printf("{");
  printf("int res = 0;");
  for(i = 0; i < size; i++)
  {
    printf("res += %d * v[%d];",u[i],i);
  }
  printf("return res;");
  printf("}");
}
```

Figure 3: Compile-time generating extension

```
int dotprod_res(int v[])
{
  int res = 0;
  res += 7 * v[0];
  res += 4 * v[1];
  res += 6 * v[2]
  return res;
}
```

Figure 4: Compile-time specialization

static computation. Furthermore, for the generation process — which must be as fast as possible at run time —, it is less expensive to directly produce "`res = 0;`" in the specialized code rather than first *lift* the value 0 (i.e., turn it into text in the "template language") and then insert it in the hole of template "`res = □;`". Because the constant is present at template compilation time rather than specialization time, the resulting binary code can also be more efficient (e.g., multiplication of a dynamic value by a literal integer can be turned into bit shifts). This binding-time feature will be used later when Tempo is applied iteratively (cf. Section 3.3). More features (not shown in this example) related to static pointer lifting and modular specialization will also be necessary.

An example of a specialized version of `dotprod` produced by `dotprod_ctgen` and invoked with actual values `size=3` and `u[]={7,4,6}` is shown in Figure 4.

3.2 Two-Level Run-Time Specialization

Tempo's strategy for run-time specialization relies on generating extensions. The difference with compile-time generating extensions is that templates are now binary rather than textual (source code). For this, source templates are pre-compiled into *binary templates*. The resulting fragments are assembled at specialization time. Holes in the binary code are patched (i.e., filled) similarly.

To obtain those binary templates, Tempo generates an extra file which defines the function `dotprod_temp` given in Figure 5. The code of this function is structured so that a standard C compiler can process the templates. This strategy contrasts with other approaches to run-time specialization which require a special-purpose compiler to be developed [3, 19].

```
int dotprod_temp(int v[])                  T1
{
  int res = 0;

  for(;dummy test;) {

    res += H1 * v[H2];                     T2[H1, H2]

  }

  return res;                              T3
}
```

Figure 5: Templates

```
dotprod_gen(int size, int u[])
{
  char *buf, *bufp;
  int i;
  bufp = buf = rts_buf_alloc();
  dump_template(bufp,t+t1,s1);
  bufp += s1;
  for(i = 0; i < size; i++)
  {
    dump_template(bufp,t+t2,s2);
    patch_hole(bufp+h1,u[i]);
    patch_hole(bufp+h2,i);
    bufp += s2;
  }
  dump_template(bufp,t+t3,s3);
  bufp += s3;
  return buf;
}
```

Figure 6: Run-time generating extension

For example, there are three templates in function `dotprod_temp` which correspond to code regions T_1, T_2 and T_3. Template T_2 contains two holes H_1 and H_2, that can be patched later at specialization time. A dummy loop has been inserted to instruct the compiler that code fragment T_2 can be executed many times. Compiling this function and performing some surgery on the binary code gives access to binary template delimitation and hole locations. (See [23, 24] for implementation details.)

Tempo then builds a run-time generating extension `dotprod_gen` that manipulates these code templates, as shown in Figure 6. In this figure, the symbol t stands for the function pointer `dotprod_temp` (an address known at load time); t_i is the offset of template T_i in t (a known integer constant); s_i is the size of the corresponding T_i templates (a known integer constant); h_j symbols are offsets of holes inside the templates (known integer constants). Operation `dump_template` (actually `memcpy`) copies the template into the specialization

```
dotprod_gen(int size, int u[])
{
  char *buf, *bufp;
  int i;
  bufp = buf = rts_buf_alloc();
  dump_template(bufp,t+t1,s1);
  bufp += s1;
  for(i = 0; i < size; i++)
  {
    dump_template(bufp,t+t2,s2);
    patch_hole(bufp+h1,u[i]);
    patch_hole(bufp+h2,i);
    bufp += s2;
  }
  dump_template(bufp,t+t3,s3);
  bufp += s3;
  return buf;
}
```

Figure 7: BTA with dynamic buffer allocation (1)

```
dotprod_gen_temp'(int u[])
{
  char *buf, *bufp;
  bufp = buf = rts_buf_alloc();            T'1[H'1]
  dump_template(bufp,H'1,s1);
  bufp += s1;

  for(;dummy test;) {

    dump_template(bufp,H'2,s2);
    patch_hole(bufp+h1,u[H'3]);             T'2[H'2, H'3, H'4]
    patch_hole(bufp+h2,H'4);
    bufp += s2;

  }

  dump_template(bufp,H'5,s3);
  bufp += s3;                               T'3[H'5]
  return buf;
}
```

Figure 8: Multi-level templates (1)

buffer. Operation `patch_hole` (actually a macro) stores a value in the buffer at a specific hole offset; this operation is processor-dependent.

Running `dotprod_gen` with actual values `size`=3 and `u[]={7,4,6}` allocates the buffer `buf` and fills it as follows. (Filling a hole in a template T with the result v of a static expression is noted $T[v]$.)

$$buf \rightarrow \boxed{T_1} \boxed{T_2[7,0]} \boxed{T_2[4,1]} \boxed{T_2[6,2]} \boxed{T_3}$$

At the end, the function returns a pointer to the beginning of the buffer where binary templates have been assembled.

3.3 Iterated Run-Time Specialization

Now assume that arguments `size`, `u[]` and `v[]` of function `dotprod` are available in this order at successive stages. We want first to specialize with respect to the size, and then with respect to a given vector. As seen in Section 2, such an incremental specialization can be obtained by specializing the generating extension `dotprod_gen` with respect to argument `size`. This requires running the binding-time analysis on the generating extension; the resulting binding times are presented in Figure 7.

As was the case for the static value `0` in the dynamic assignment "`res = 0;`" of Figure 2, the analysis does not treat the literal integers h_i and s_i as static because they are in a dynamic context. Without this optimization, incremental specialization would incur the cost of a useless hole-patching for *each* code generation operation (template dumping and hole filling) in the first generating extension.

Tempo's BTA also has a special treatment of static pointers. It considers static the pointer expressions t+t_i because t is an address known at load time and integers t_i are literal constants. However, these expressions occur in a dynamic context (i.e., a call to `dump_template`).

In the case of compile-time specialization, static pointer values cannot be lifted (i.e., translated into a textual representation in the specialized code) when they are in a dynamic context. Indeed, pointer values can vary from one execution to another; moreover, pointed values might have to be lifted as well. Thus, such pointer expressions are usually turned into dynamic by the BTA [15]. As mentioned above, because there is no need to consider literal constants as static when they are in a dynamic context, this would actually result in the binding time "`dump_template(bufp,`t+t_i`,`s_i`);`".

Fortunately, there is no such constraint in the context of run-time specialization. Indeed, lifting a static pointer is just like lifting an integer, i.e., the identity function. However, the lifetime of the pointed memory space should be considered carefully: not only should it exist at specialization time but it should also exist at execution time. This condition is not guaranteed for pointers to heap-allocated memory (that can be freed) nor stack-allocated memory (that becomes invalid when the corresponding function returns; moreover, the address of local variables can vary from one call to another). In order for pointer lifting to be safe, it is enough to restrict this operation to global locations. Tempo implements such a feature, and thus may produce different binding-time annotations depending on the type of specialization chosen (compile-time or run-time). In our example, the BTA produces the annotation "`dump_template(bufp,`t+t_i`,`s_i`);`". The improvement over a traditional BTA is small in this case: `size`+2 additions are now performed at specialization time rather than execution time. However, this feature will be crucial when further optimizing the incremental specialization process (cf. Section 3.4).

The binding-time analyzed version of the generat-

```
dotprod_gen_gen′(int size)
{
  char *buf′, *bufp′;
  int i;
  bufp′ = buf′ = rts_buf_alloc();
  dump_template(bufp′,t′+t′1,s′1);
  patch_hole(bufp′+h′1,t+t1);
  bufp′ += s′1;
  for(i = 0; i < size; i++)
  {
    dump_template(bufp′,t′+t′2,s′2);
    patch_hole(bufp′+h′2,t+t2);
    patch_hole(bufp′+h′3,i);
    patch_hole(bufp′+h′4,i);
    bufp′ += s′2;
  }
  dump_template(bufp′,t′+t′3,s′3);
  patch_hole(bufp′+h′5,t+t3);
  bufp′ += s′3;
  return buf′;
}
```

Figure 9: Multi-level run-time generating extension (1)

ing extension leads to templates shown in Figure 8. These templates contain operations depending on the data available at this stage, e.g., u[H'_3]. They also contain template management of the previous stage. From this binding-time analysis is also produced the three-level generating extension shown in Figure 9.

The overall behavior is the following. Running the function dotprod_gen_gen′ with the actual value 3 for size allocates a first buffer for the intermediate specialization; it then loads it with templates T'_i and fills them.

$$buf' \rightarrow \boxed{T'_1[t{+}t_1]}\boxed{T'_2[t{+}t_2,0,0]}\boxed{T'_2[t{+}t_2,1,1]}\ \boxed{T'_2[t{+}t_2,2,2]}\boxed{T'_3[t{+}t_3]}$$

Then, running the function at address buf' on a given u[] produces the same effect as dotprod_gen as described earlier in Section 2.3.

Note that each time the code at buf' is run, a new buffer buf is allocated. Hence, many specializations with respect to size and u[] may coexist. However, there are cases where this is not needed. For example, in the case of our loop nest program, the uses of function f_gen_res are not simultaneous but successive (for each j). If we know that coexisting specializations are not needed at a given stage, further optimization can be achieved, as described in the next section.

3.4 Optimized Iterated Specialization

Assuming that coexisting specializations with respect to u[] are not needed, we may allocate a single specialization buffer for each given size. This amounts to considering the allocation of the specialization buffer in dotprod_gen as static rather than dynamic. This information can be exploited to further factorize the specialization process. In particular, calls to dump_template can now be performed at the first specialization stage. The resulting binding-time analyzed program is shown in Figure 10. As can be noticed, compared to the analyzed program in Figure 7, many more computations have been made static.

```
dotprod_gen(int size, int u[])
{
  char *buf,  *bufp;
  int i;
  bufp = buf = rts_buf_alloc();
  dump_template(bufp,t+t1,s1);
  bufp += s1;
  for(i = 0; i < size; i++)
  {
    dump_template(bufp,t+t2,s2);
    patch_hole(bufp+h1,u[i]);
    patch_hole(bufp+h2,i);
    bufp += s2;
  }
  dump_template(bufp,t+t3,s3);
  bufp += s3;
  return buf;
}
```

Figure 10: BTA with static buffer allocation (2)

It must be noted that expression bufp+h_1 is a static pointer in a dynamic context, as was the case for expressions t+t_i in Section 3.3. If static pointers in dynamic contexts could not be lifted, they would have to be dynamic. Then the initial buffer allocation as well as calls to dump_template would have to be residualized. This would lead to binding times similar to those in Figure 7. Thus, it would not be possible to exploit the static buffer allocation.

To specify that the buffer allocation should be static, we rely on Tempo's support for *modular specialization*, i.e., the ability to specialize only a part of a program. In Tempo, a model of the operational behavior can be specified for all external functions. In our case, we model rts_buf_alloc as a function returning a constant global pointer. It is also possible to specify if external functions can be called at specialization time, provided they do not contain any dynamic fragment. We thus declare function rts_buf_alloc as executable at specialization time.

The combination of all these features are required to obtain the binding times in Figure 10. The only dynamic action is the patch of the values u[i] at given addresses. Compared to the dynamic buffer allocation case, corresponding templates are much smaller and simpler, as shown in Figure 11. The resulting three-level generating extension is shown in Figure 12.

The overall behavior is the following. Running the function dotprod_gen_gen″ with the actual value 3 for size generates two specialization buffers in a row and

```
dotprod_gen_temp''(int u[])                T1''
{

  for(;dummy test;) {

    patch_hole(H1'',u[H2'']);              T2''[H1'', H2'']

  }

  return H3'';                             T3''[H3'']
}
```

Figure 11: Multi-level templates (2)

```
dotprod_gen_gen''(int size)
{
  char *buf'', *bufp'';
  char *buf, *bufp;
  int i;
  bufp'' = buf'' = rts_buf_alloc();
  dump_template(bufp'',t''+t1'',s1'');
  bufp'' += s1'';
  bufp = buf = rts_buf_alloc();
  dump_template(bufp,t+t1,s1);
  bufp += s1;
  for(i = 0; i < size; i++)
  {
    dump_template(bufp,t+t2,s2);
    dump_template(bufp'',t''+t2'',s2'');
    patch_hole(bufp''+h1'',bufp+h1);
    patch_hole(bufp''+h2'',i);
    bufp'' += s2'';
    patch_hole(bufp+h2,i);
    bufp += s2;
  }
  dump_template(bufp,t+t3,s3);
  bufp += s3;
  dump_template(bufp'',t''+t3'',s3'');
  patch_hole(bufp''+h3'',buf);
  bufp'' += s3'';
  return buf'';
}
```

Figure 12: Multi-level run-time generating extension (2)

loads them both with pre-filled templates T_i and T_i''. Values a_i are pre-computed addresses $buf + s_1 + i \times s_2 + h_1$; they are the addresses of the three holes in buf (denoted by "□").

$buf \rightarrow$	T_1	$T_2[\square, 0]$	$T_2[\square, 1]$	$T_2[\square, 2]$	T_3
$buf'' \rightarrow$	T_1''	$T_2''[a_0, 0]$	$T_2''[a_1, 1]$	$T_2''[a_2, 2]$	$T_3''[buf]$

This completes the first specialization stage. Then, running the function at address buf'' merely amounts to filling the three holes of buf at address a_i with `u[i]` values; the function at address buf is then ready for execution.

Function	Time
`dotprod`	1.67
`dotprod_res` (compile-time)	0.54
`dotprod_res` (run-time)	0.87
`dotprod_gen` (RT dyn alloc)	24.0
`dotprod_gen` (RT stat alloc)	17.6

Table 1: Traditional specialization of `dotprod`

Function	Dynamic allocation	Static allocation
`dotprod_res`	0.87	0.87
`dotprod_gen_res`	22.8	1.96
`dotprod_gen_gen`	81.1	96.4

Table 2: Incremental specialization of `dotprod`

3.5 Experiment

We ran a preliminary experiment with incremental run-time specialization on our `dotprod` example. For this, we used a Sparc Ultra 1 / 170 MHz running SunOS 5.5. All files were compiled by `gcc` with optimization option `-O1`. Execution times for traditional (run-time and compile-time) and incremental specialization are provided respectively in Table 1 and Table 2. All times are given in seconds for one million of calls, for a vector `size` of 10 and any known vector `u[]`. (Actual values of `u[]` have no impact except, possibly, at compile time.) Figures are given for both static and dynamic buffer allocations. When memory allocation is involved, the time also includes freeing the allocated memory. Due to the very small running time of all the tested functions, the figures should be considered with caution.

These results show that all versions of `dotprod_res` have the same execution time. That is coherent with the fact that they are all identical.

For traditional specialization, the specialized function is twice as fast as the original code when the buffer allocation is dynamic, and 30 executions are needed for specialization to pay off. When the buffer is static, the specialized function is 3 times faster and specialization is amortized after 22 executions.

In the incremental case, as soon as the vector size is known, only 2.5 to 28.5 uses of the specialized function (whether the specialization buffer is static or not) are required to make it profitable. It is important to note that, in the static buffer case, incrementality makes the specialization break-even point 9 times smaller than for traditional specialization. As can be expected, static buffer allocation is better than dynamic allocation as soon as the first use.

4 Discussion

In this section, we analyze our iterative specialization approach. The following section compares it with related work.

4.1 Degree of Specialization

An important issue about incremental specialization is whether the iterated approach has an impact on the degree of specialization. Since "native" multi-level specialization is specially defined to achieve incremental specialization, the question actually is: does iterated specialization lose any specialization opportunity?

It is difficult to make a general statement on this issue because an answer requires thoroughly formalizing the two models (for a given language, a given BTA precision, a given code generation mechanism, etc.). What we try here is to provide informal evidence that, given some assumptions, the two approaches are equivalent as far as the exploitation of data available at each stage is concerned.

Obviously, to achieve iterated specialization, a partial evaluator must first of all be able to handle the language constructs that are used by the generating extensions that it produces. Then comes the issue of binding times, which express the degree of specialization. As illustrated by the examples in Figures 3 and 6, a generating extension contains two intertwined pieces of code: one that is a copy of the static slice of the original program (it also defines the overall control of the specialization), and one that manages code generation (buffer allocation, template assembling, hole filling). When further staging the static arguments, the BTA must exploit the specialization opportunities offered by the availability of more data without being disrupted by code generation. Since template management does not affect the control flow and operates on separate memory states, the only possible interference is through the data that is exchanged with the static slice, i.e., the computed values that are put into template holes. In our case, this corresponds to the last argument of `patch_hole`. There are four cases to examine: whether this argument is static or dynamic in the new binding-time division of the previous static stage (e.g., expressions "`i`" and "`u[i]`"), and whether the hole filling operation is forced to dynamic or can be static (e.g., see Figures 7 and 10). The only possible impact is when the hole filling operation is dynamic whereas the value to put into the hole is static (e.g., "`patch_hole(bufp+`h_2`,i);`"): because it is in a dynamic context, the static expression should be turned dynamic by the BTA if it cannot be lifted. Yet, we know that this expression must necessarily be liftable since it is the argument of a hole filling operation: this means that it was already a static expression in a dynamic context in the previous binding-time stage, which thus resulted in a template with a hole to be filled.

From this informal reasoning, we can conclude that there is no interference between multi-level static computations and code generation. Iterated specialization exploits as much specialization opportunities as multi-level specialization.

4.2 Engineering Effort

Iterated run-time specialization is simple. As can be seen in Section 3, there is no need to turn the first template object file and corresponding pointers into textual data in order to apply specialization a second time. The actual values of template addresses are determined at load time and thus available at run time. All template object files, as well as the second-iteration generating extension are linked together into a single file.

We implemented our incremental specialization process in Tempo almost "for free". We only had to make very minor changes, mainly to prevent name clashes and multiple definitions when building a second-iteration generating extension of an already produced generating extension. All the other features that we used (modular specialization, static pointers lifting and dynamic literal constants) had already been implemented in Tempo for other applications.

Iterated specialization requires applying partial evaluation $n-1$ times if n stages are required. This can be laborious although part of it could be automated. However, besides loop nests, for all applications we have considered so far, the number of levels of incremental specialization is actually equal to three, thus requiring only two applications of a partial evaluator. Yet, in principle, a multi-level BTA should be able to process a program more efficiently than our iterated process because of its global knowledge of the stages. The iterated process only processes two levels at a time. There are redundancies in the determination of binding times at each stepwise refinement: although staged later in following iterations, computations are first determined static as a whole.

Besides, iterated specialization allows incremental partial evaluation to be tuned for each stage (e.g., static or dynamic buffer allocation). Even if a similar functionality could easily be defined for multi-level specialization, developing a multi-level specializer, when a two-level specializer is already available, does not seem worth the effort; this is even more so at run time because run-time code generation requires complex back-ends.

5 Related Work

There is an obvious relationship between incremental partial evaluation in loop nests and code motion of loop invariants as found in optimizing compilers [1, 21].

The difference is that incremental partial evaluation can handle any type of invariants (structures, arrays, pointers), not only scalars. Moreover, incremental partial evaluation factorizes computations inter-procedurally, whereas code motion in compilers is usually only intra-procedural. Autrey and Wolfe proposed a staging analysis, named *glacial variable analysis*, aimed at detecting variables in loop nests that are good candidate for incremental run-time specialization [4].

There exists a variety of code generation strategies, depending on the target language and the specialization time (before compiling or while running the program). Since incremental specialization only amounts to optimizing the specialization process, the speed of the code generation process is a crucial issue for realistic applications.

Incremental specialization has been proposed for functional languages [13, 33]. Because this work is limited to compile time, a comparison with our approach is difficult. Indeed, when performing compile-time specialization, the code generation process is not optimized for speed.

There exist other run-time code generation systems, but reports on these systems do not mention any support for incremental specialization. The Fabius system compiles a pure, first-order subset of ML into native MIPS code [19]. Some issues like register allocation are decided at compile time whereas instruction selection is performed at run time. The Tick C compiler generates code at run time from a C program where computations are explicitly staged using Lisp-like backquote notations [9, 27]. The DyC system compiles partially annotated C programs. Like Tempo, it produces templates which are compiled by the DEC Alpha compiler [3, 14]. Unlike Tempo, it performs additional optimizations that can exploit template instantiation values and template assembly. Data are not yet available to assess the impact of these optimizations.

$ML^{\Box}$ performs incremental run-time code generation but do not produce native code. It compiles a subset of ML augmented with specific code generation constructs into the CCAM, an extension of the Categorical Abstract Machine [35]. The consistency of the code generation constructs are checked by the compiler. In contrast, Tempo offers automatic staging based on annotations produced by a binding-time analysis. Benchmarks on $ML^{\Box}$ are reported in terms of a number of reductions steps in the CCAM. Besides $ML^{\Box}$, a run-time code generation system for Scheme has been constructed by composing a partial evaluator and a bytecode compiler [30]; incremental specialization should be possible with this system. Yet, as for $ML^{\Box}$, the nature of the source language and the target code makes the comparison with our work difficult.

6 Future Work

Incremental run-time specialization aims at making specialization (i.e., code generation) faster. There is a tradeoff between the quality of the generated code and the speed to produce it. Finely tuning this tradeoff is important for the practical use of incremental specialization. Code generation in Tempo is currently very fast. However, Tempo does not perform any inter-template optimization, nor does it take advantage of specific values that are put into template holes. To improve the quality of the code, we are investigating the development a dynamic peephole optimizer. Implementation of run-time inlining when specializing a function is in progress.

Besides, we are also considering source-level transformations to encode optimizations to be performed at specialization time by the specializer itself; this includes some cases of algebraic simplifications and strength reduction. Source-level transformations can also cache determined memory cells into local variables, that are compiled more efficiently into machine registers. Assuming this caching is static, specialization is a little slower because it compiles the caching process, but the specialized function is faster because it makes less accesses to memory.

Beside techniques, we are also considering applications. We are investigating the development of a generic virtual machine for mobile bytecode [10]. The idea is to parameterize this generic virtual machine with respect to both a definition of bytecode instructions and a bytecode program. The mobile nature of the application makes it critical to use run-time specialization. Furthermore, it is likely that a bytecode definition will apply for a series of bytecode programs. This situation creates a need to factorize the specialization of the generic virtual machine with respect to a given bytecode definition. The goal is to achieve fast, efficient on-the-fly compilation like a just-in-time compiler.

7 Conclusion

We have presented an approach to incremental run-time specialization which allows programs to be optimized at several stages, as data become available.

The main advantage of incrementality is to factorize the specialization phase: instead of specializing a program all at once, as is traditionally done, incremental specialization allows this process to be staged. As a result, specializing a program at a given stage costs considerably less than specializing it once all the data are available. In addition, according to the number of simultaneous uses of a specialized function at a given stage, we have shown how to further optimize the incremental specialization process. We have described how incremental run-time specialization can be achieved using an existing partial evaluator. Our approach is implemented in a program specializer for C named Tempo.

Although our preliminary experiment is encouraging, realistic applications are now necessary to validate the approach.

Acknowledgment

We would like to thank Franois Nol for early discussions and ideas on this topic, and for testing the feasibility of the approach by making preliminary experiments in and with the run-time specializer of Tempo. Julia Lawall also provided helpful comments on this paper.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] A. Aiken, M. Fähndrich, J. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Second International Workshop on Types in Compilation (TIC '98)*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[3] J. Auslander, M. Philipose, C. Chambers, S.J. Eggers, and B.N. Bershad. Fast, effective dynamic compilation. In PLDI'96 [25], pages 149–159.

[4] T. Autrey and M. Wolfe. Initial results for glacial variable analysis. In *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing (LCPC), Santa Clara, California*, volume 1239 of *Lecture Notes in Computer Science*. Springer-Verlag, August 1996.

[5] C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, and N. Volanschi. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys, Symposium on Partial Evaluation*, 30(3), 1998.

[6] C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, February 1996.

[7] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In POPL96 [28], pages 145–156.

[8] C. Consel, C. Pu, and J. Walpole. Incremental specialization: The key to high performance, modularity and portability in operating systems. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 44–46, Copenhagen, Denmark, June 1993. ACM Press. Invited paper.

[9] D.R. Engler, W.C. Hsieh, and M.F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In POPL96 [28], pages 131–144.

[10] Bertil Folliot, Ian Piumarta, and Fabio Riccardi. A dynamically configurable, multi-language execution platform. In *Eighth ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, September 1998.

[11] R. Glück. Towards multiple self-application. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 309–320, New Haven, CT, USA, September 1991. ACM SIGPLAN Notices, 26(9).

[12] R. Glück and J. Jrgensen. Efficient multi-level generating extensions for program specialization. In M. Hermenegildo and S. Doaitse Swierstra, editors, *Proceedings of the 7^{th} International Symposium on Programming Language Implementation and Logic Programming*, number 982 in Lecture Notes in Computer Science, pages 259–278, Utrecht, The Netherlands, September 1995.

[13] R. Glück and J. Jrgensen. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation*, 10:113–158, 1997.

[14] B. Grant, M. Mock, M. Philipose, C. Chambers, and S.J. Eggers. Annotation-directed run-time specialization in C. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 163–178, Amsterdam, The Netherlands, June 1997. ACM Press.

[15] L. Hornof, J. Noyé, and C. Consel. Effective specialization of realistic programs via use sensitivity. In P. Van Hentenryck, editor, *Proceedings of the Fourth International Symposium on Static Analysis, SAS'97*, volume 1302 of *Lecture Notes in Computer Science*, pages 293–314, Paris, France, September 1997. Springer-Verlag.

[16] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* International Series in Computer Science. Prentice-Hall, June 1993.

[17] N.D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, volume 202 of *Lecture Notes in Computer Science*, pages 124–140. Springer-Verlag, 1985.

[18] J.L. Lawall. Faster Fourier transforms via automatic program specialization. Publication interne 1192, IRISA, Rennes, France, May 1998.

[19] P. Lee and M. Leone. Optimizing ML with run-time code generation. In PLDI'96 [25], pages 137–148.

[20] R. Marlet, S. Thibault, and C. Consel. Mapping software architectures to efficient implementations via partial evaluation. In *Conference on Automated Software Engineering*, pages 183–192, Lake Tahoe, Nevada, November 1997. IEEE Computer Society.

[21] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers, 1997.

[22] G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 240–249, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

[23] F. Noël. *Spécialisation dynamique de code par évaluation partielle.* PhD thesis, Université de Rennes I, October 1996. In French.

[24] F. Noël, L. Hornof, C. Consel, and J. Lawall. Automatic, template-based run-time specialization : Implementation and experimental study. In *International Conference on Computer Languages*, pages 132–142, Chicago, IL, May 1998. IEEE Computer Society Press. Also available as IRISA report PI-1065.

[25] *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, PA, May 1996. ACM SIGPLAN Notices, 31(5).

[26] *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, Nevada, June 15–18, 1997.

[27] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. `tcc`: A system for fast, flexible, and high-level dynamic code generation. In PLDI'97 [26], pages 109–121.

[28] *Conference Record of the* 23^{rd} *Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, St. Petersburg Beach, FL, USA, January 1996. ACM Press.

[29] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the 1995 ACM Symposium on Operating Systems Principles*, pages 314–324, Copper Mountain Resort, CO, USA, December 1995. ACM Operating Systems Reviews, 29(5), ACM Press.

[30] Michael Sperber and Peter Thiemann. Two for the price of one: Composing partial evaluation and compilation. In PLDI'97 [26], pages 215–225.

[31] S. Thibault, C. Consel, and G. Muller. Safe and efficient active network programming. In *17th IEEE Symposium on Reliable Distributed Systems*, pages 135–143, West Lafayette, Indiana, October 1998.

[32] S. Thibault, R. Marlet, and C. Consel. A domain-specific language for video device drivers: from design to implementation. In *Conference on Domain Specific Languages*, pages 11–26, Santa Barbara, CA, October 1997. Usenix.

[33] Peter J. Thiemann. Cogen in six lines. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 180–189, Philadelphia, Pennsylvania, 24–26 May 1996.

[34] E. N. Volanschi. *Une approche automatique à la spécialisation de composants système.* Thèse de doctorat, Université de Rennes I, February 1998.

[35] Philip Wickline, Peter Lee, and Frank Pfenning. Run-time code generation and modal-ml. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 224–235, Montreal, Canada, 17–19 June 1998.

[36] Kwangkeun Yi and Williams Ludwell Harrison III. Automatic generation and management of interprocedural program analyses. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 246–259, Charleston, SC, USA, January 1993. ACM Press.