Tailored Termination for Optimal Supercompilation

Ammar Askar
Purdue University
aaskar@purdue.edu

Benjamin Delaware
Purdue University
bendy@purdue.edu

Abstract

Supercompilation is a simple, powerful program specialization technique, but it can face issues with tractability and code bloat. In this paper, we examine how the choice of the termination criteria, one of the key components of the supercompilation algorithm, can effect the performance of supercompiled programs. We have conducted an empirical study of the effect of selecting a termination criteria which is optimal with respect to the program being specialized. We observe that such a tailored criteria can yield significant improvements over a one-size-fits-all criteria, with no potential for performance degradation. While our study focuses on user-supplied termination criteria, we theorize how this tailoring process might be automated.

Keywords Supercompilation, Termination, Performance, Haskell

1 Introduction

Supercompilation [22] is a powerful program specialization technique which can produce programs that are considerably more efficient than the original [12]. The source of the technique's power is its ability to drive arbitrarily deeply into a program's execution during specialization. This allows supercompilation to subsume many manually implemented compiler optimizations including deforestation, function specialization, and constructor specialization [2, 20]. Unfortunately, this power cuts both ways: left unchecked, the supercompilation algorithm can specialize a program ad infinitum. Thus, a key component of any supercompiler is the *termination criteria* it uses to halt specialization. The choice of termination criteria is critical to the performance of the specialized program: potential improvements can be unrealized if supercompilation halts too soon; and overly permissive criteria can result in a bloated, overspecialized program that is actually *less* performant [15].

The hypothesis of this paper is that rather than applying a single, one-size-fits-all termination criteria, we can avoid both under and overspecialization by selecting the best termination criteria *for the program being specialized*. To test this idea, we have developed a termination checker that enables precise, per-program termination criteria, and empirically measured the performance of such criteria on supercompilation for over 30 benchmarks. Our preliminary results are encouraging: on average, we can achieve an 8% improvement in runtime, and an 18% decrease in memory allocated over the original program, with no performance degradation in the worst case. While our experiments rely on a user to select an optimal criteria, we hypothesize that this approach can be automated through either profiling or machine learning.

The rest of the paper proceeds as follows: we begin by discussing the effects of different termination criteria on supercompilation. Next, we explain our approach to programmer-specified termination criteria, which we then use to perform an empirical comparison of tailored termination criteria against the fixed termination criteria of Bolingbroke et al. [2]. We finish with a discussion of related work and potential future directions for tailored termination criteria.

2 Termination Criteria

One common form of a termination criteria is the check used by an optimizing compiler to avoid repeatedly inlining recursive functions; these typically utilize heuristics based on program size [1]. Supercompilers require a similar check to avoid infinite loops during specialization. Consider a program that sums up all the elements of two lists: $\lambda xs.\ \mathit{fold}\ (+)\ 0\ ([1,2] \mathbin{++} xs)$. Without a termination check, the supercompiler would repeatedly inline the *fold*, leading to an infinite loop.

The choice of termination criteria can have considerable impact on the performance of specialized code. As an example, consider the following program:

$$
\begin{aligned}
&\textbf{let}\ \mathit{longId} = \lambda n.\lambda x.\textbf{if}\ n == 0\\
&\qquad\qquad\qquad\qquad \textbf{then}\ x\\
&\qquad\qquad\qquad\qquad \textbf{else}\ \mathit{longId}\ (n-1)\ x\\
&\textbf{in}\ \mathit{longId}\ 100\ y
\end{aligned}
$$

This contrived function is an identity function that returns its second argument when the accumulator argument hits zero. Thus, the ideal specialization of *longId 100 y* would be y. To see how an overly conservative termination criteria can hinder optimization, consider what happens during the supercompilation of this program. A supercompiler would perform as many runtime reductions at compile time as possible. The program after one such reduction step is:

$$
\begin{aligned}
&\textbf{let}\ \mathit{longId} = \ldots\\
&\textbf{in}\ \textbf{if}\ 100 == 0\\
&\quad \textbf{then}\ x\\
&\quad \textbf{else}\ \mathit{longId}\ (100-1)\ x
\end{aligned}
$$

The next step reduces the **if** statement to produce:

$$\textbf{let}\ longId = \ldots$$
$$\textbf{in}\ longId\ (100-1)\ y$$

After 99 additional sequences of reductions, we would reach the optimal program, y. Based on the growth in program size, however, most termination criteria would estimate that this sequence of reductions is diverging and halt supercompilation before the ideal program is found. Therein lies the crux of the problem: general termination algorithms will have trouble identifying that *longId* will eventually be completely specialized away. The next section presents a way to loop a programmer into the supercompilation process in order to define a better criteria.

3 Custom Termination Criteria

Our experiments with custom termination criteria are based on the supercompiler of Bolingbroke et al. [2, 3], which is parameterized over a termination check. Their supercompilation algorithm is:

```
supercompile :: History → State → State
supercompile hist state = case terminate hist state of
    Continue hist'   → let state' = evalStep state in
                         split (supercompile hist' state')
    Stop             → state
```

The three main components of the algorithm are the *evalStep* function, which takes a particular state and performs a reduction step on the term being scrutinized; the *split* function, which tries to split a stuck term into subterms, such as splitting $y :: [String]$ into $(x : xs)$ and [] for further reduction; and the *terminate* function that decides whether the algorithm should halt. The algorithm creates a history of previously seen states that *terminate* uses to make its decision. The signature of *terminate* is:

```
terminate :: History → State → TermRes
data TermRes = Stop | Continue History
```

A *sound* termination criteria is one that, when given a list of states $s_1, s_2, s_3, s_4, \ldots s_n$, will eventually return a signal of *Stop* at some state s_i. A sound criteria thus guarantees that *supercompile* will eventually halt.

The most conservative implementation of *terminate* immediately signals stop, resulting in an unchanged program. The least conservative (and an unsound) implementation of *terminate* always returns a continue signal:

```
terminateImmediately _ _ = Stop
terminateNever _ _ = Continue emptyHistory
```

It is important to note that previous works that have attempted to prove what termination criteria are sound consider soundness for *all* possible programs [4]. While a termination check might not be sound in general, it may be perfectly reasonable for a certain subset of programs. For example, the *terminateNever* criterion above is unsound in general, but it halts for the *longId* example in section 2.

The termination check is not the only mechanism for halting supercompilation— a program blocked on free variables is also considered to be done compiling, without the need for *terminate* to signal a stop. As an example, consider:

$$\textbf{let}\ sum\ x\ y = x + y\ \ \textbf{in}\ \ sum\ a\ b$$

Supercompilation performs a β reduction to produce $a + b$, but is then blocked at the next step, as no reductions can be performed without knowledge of the free variables.

Equally important to effective supercompilation is avoiding overspecialization [5, 21], which can cause the supercompiler to slow down as it explores a potentially exponential number of program states. In addition, it can lead to larger programs, which tend to perform poorly when compared to their shorter counterparts. The sheer number of instructions in the bloated compiled programs can create a large number of cache misses [9] and make it harder for lower level compiler passes to optimize them. Techniques like supercompiler rollback [2] attempt to curb this issue, but as section 2 discussed, a globally defined heuristic based on, e.g. code size, can leave optimization opportunities on the table.

It is reasonable to ask then, how we might define an optimal termination criteria on a per-program basis. To answer this question, we first consider an implementation of *terminate* that can be easily stopped at an *arbitrary* specialization point. Uniquely tagging each state in a history allows users to precisely specify when to terminate by identifying the set of terminal states. These tags can then be fed back into the supercompiler in order to terminate at an optimal state. This new termination criteria looks like:

```
tag :: History → State → Tag
terminalTags :: [Tag]

terminateOnTag history state =
    if elem (tag (history state)) terminalTags
    then Stop
    else Continue emptyHistory
```

The next section explains how states are tagged via an example, and section 4 discusses the merits of the different ways to present the state tags to the users.

3.1 User-Specified Termination By Example

To illustrate how a user can specify termination, we work through an example specialization of the *fold* example from section 2.:

```
let fold = λc n xs. case xs of [] → []
                              (x : xs) → fold c (c n x) xs
in λxs. fold (+) 0 ([1, 2] ++ xs)
```

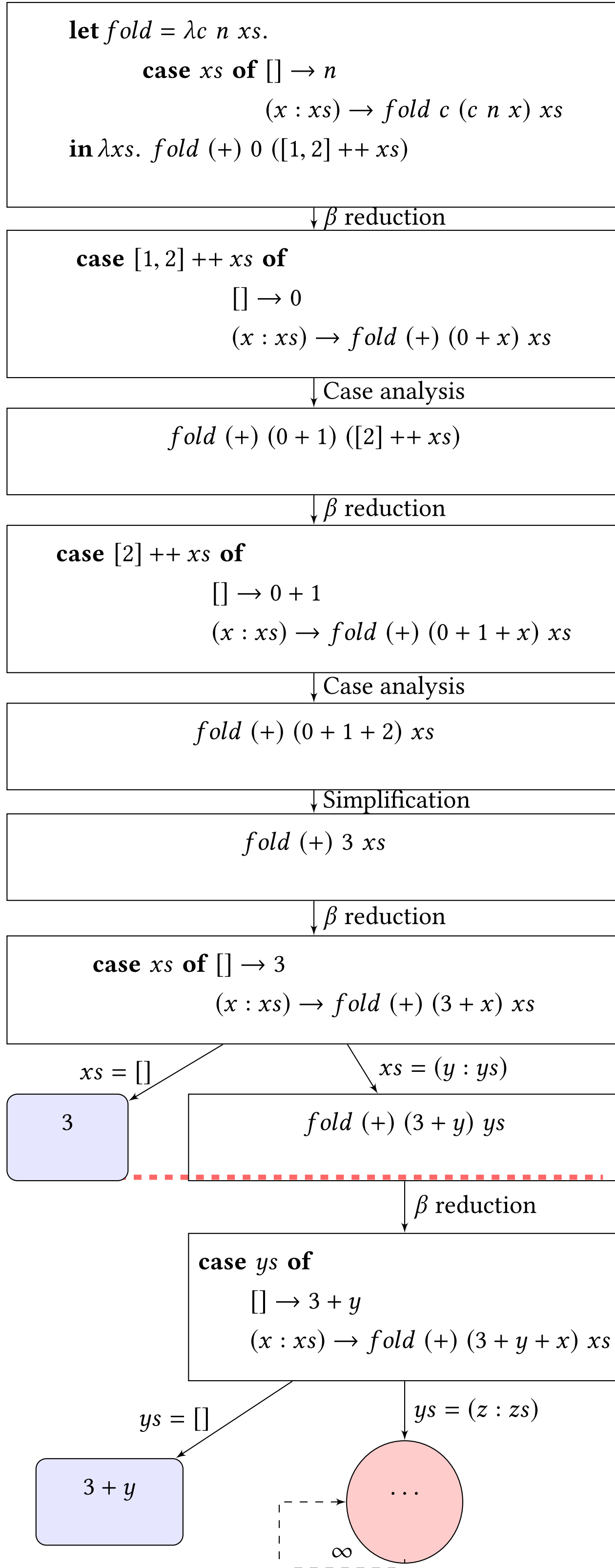

Figure 1. Visualization of supercompilation not terminating

The first two elements in the list are constants and can thus be processed at compile time to produce an optimized version of this program:

$$\begin{aligned}&\textbf{let}\ fold = \ldots\\&\textbf{in}\ \lambda xs.\ fold\ (+)\ 3\ xs\end{aligned}$$

Looking at Figure 1, the supercompiler reaches the desired specialization at the dashed line. Terminating supercompilation at that point and inlining all of the supercompiled states produces the following residual program:

$$\begin{aligned}&\textbf{let}\ fold = \ldots;\\&\textbf{in}\ \lambda xs.\ \textbf{case}\ xs\ \textbf{of}\ [] \rightarrow 3\\&\qquad\qquad\qquad (y : ys) \rightarrow fold\ (+)\ (3 + y)\ ys\end{aligned}$$

Continuing past the dashed line quickly leads to a nonterminating specialization pass. After beta reducing the definition of $fold$, the algorithm splits on its first argument in order to specialize the case statement.

Specializing the cons case of this case expression requires examining a recursive call to *fold*. Repeating the process of beta reducing and examining the cases again, the algorithm heads into an infinite loop. Memoization is capable of preventing some similar forms of infinite inlining, as in the case of *map* but calls to *fold* differ in each invocation, preventing memoization from being effective. Instead, by examining the state of the supercompiler a programmer can signal when a fully specialized version has been produced. In this example, this amounts to selecting the two nodes before the dashed line as the terminal tags.

3.2 Tagging

Assigning distinct tags to each state in the history is essential to giving complete control over the termination criteria. We will use our running example to detail our method for uniquely tagging supercompilation states. This method treats the history of the supercompiler as a tree whose nodes represent a state and whose edges represent reduction and splitting steps, per Figure 1. With this interpretation, finding unique tags for each state is simply a matter of uniquely labelling each to each node in this tree. Since these tree can become infinitely large, we do not aim to completely process the full tree, but instead generate labels on-the-fly for partial trees. That is, given the same tree but with new branches where leaves used to be, the labels should stay the same. This allows the supercompiler to be paused at any particular state in order for users to select intermediate states as terminal, and then safely resumed, as all the tags will remain unique. In order to satisfy these requirements, the tag for each state

is a list of numbers and our tagging algorithm is as follows:

$$
\begin{aligned}
&tag\ hist\ s = \textbf{case}\ s\ \textbf{of} \\
&\quad (Splitted\ term\ index) \rightarrow index : tag\ hist\ (prevState\ hist) \\
&\quad (Term\ term) \rightarrow \textbf{if}\ isRoot\ term \\
&\qquad\qquad \textbf{then}\ [0] \\
&\qquad\qquad \textbf{else}\ 0 : tag\ hist\ (prevState\ hist)
\end{aligned}
$$

The initial state, i.e the root term is assigned a tag of [0]. *Splitted* is the result of the *split* function with the *index* representing which sub-term case the stuck term was split using. For example, the empty list in Figure 1 might be given an index of 1, while the cons case an index of 2.

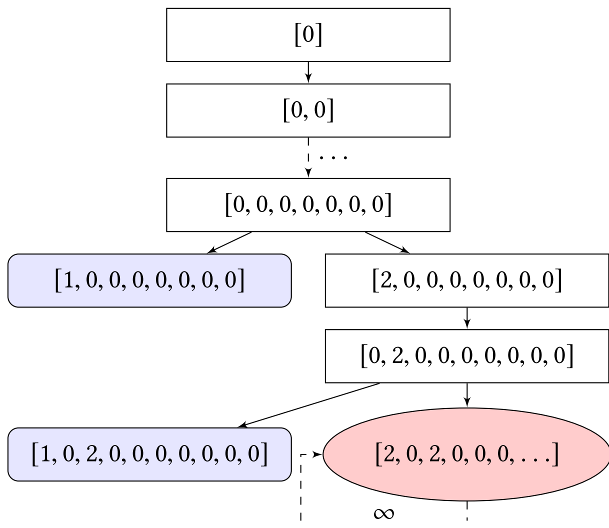

Figure 2. Tags for nodes in Figure 1.

Figure 2 shows the result of applying this algorithm to the nodes in Figure 1. Applying this algorithm to any further unrolling of this tree (from the top down) would provide the exact same labels.

4 Evaluation

We have implemented a supercompiler which supports tailored termination criteria in a fork of Max Bolinbroke's "Cambridge Haskell SuperCompiler" (*chsc*) [5]. We then evaluate the performance of our supercompiler against *chsc*, which uses a uniform tag-bag termination criteria for all programs.

Our key evaluation metrics are as follows:

- **Program run time** to measure optimization and increases in efficiency.
- **Total memory allocated** to account for the benefits of deforestation and stream-fusion like optimizations.
- **Program size** to factor in the decrease in code bloat caused by better termination.
- **Compilation time** to account for the overhead of supercompilation. Short compile times are necessary to prevent harm development efficiency.

While past supercompilers do well in the first two categories, we show that with tailored termination criteria, gains can be made in all four categories.

Benchmarks Our benchmarks are drawn from a combination of previous supercompilation papers and standard Haskell performance benchmarks. From the latter category, we used the imaginary portion of the nofib benchmark suite [18]. These benchmarks were written without consideration of deforestation or supercompilation, so they reflect the effects of supercompilation in programs without "obvious" optimization opportunities.

Next, in order to evaluate how well a supercompiler can exploit opportunities for deforestation, we used the programs outlined in Jonsson's PhD thesis on supercompilation [11]. These programs are designed with obvious ways to optimize them. For instance, the "tree" example involves a *map* function written over a custom tree data structure.

Lastly, we also use some of the original benchmarks created by Bolingbroke for his PhD thesis [5]. Some of these have obvious optimization paths, while others test how well specialization of certain partially applied functions plays out.

Methodology One straightforward method to select the optimal halting states is to simply present a user with the current state and ask them whether to continue or not. Unfortunately, this does not scale as the number of specialization states grows. Supercompilation can involve hundreds to thousands of termination checks [16]; asking a user to provide input for all such states is simply not viable. Instead, a far more scalable approach uses the generally unsound $terminateNever$ criteria from section 2. To run our experiments, we supercompile programs using this criteria, pausing compilation after a fixed number (5–20) of steps. When paused, the compiler presents its internal history to the user, who can scroll through the history to identify the point where the supercompiler began to bloat the code.

Like *chsc*, our compiler acts as a preprocessor for Haskell programs, supercompiling them before passing them into GHC. The supercompiler compiles a program as a large single unit. Hence, it requires definitions of all functions, even those from GHC's standard library (Prelude). All benchmarks are compiled with the `-O2` flag to enable GHC to perform its internal optimizations, in order to get a fairer comparison rather than against a naive compiler. All benchmarks were performed on a machine with an Intel i7-7700HQ processor and 16 GB of RAM. Following best practices [13], we provide confidence intervals [8] for speed up and slow down ratios/percentages. Effect sizes are not reported for program size and memory allocated, as we found them to be deterministic. For full transparency, we include the extra development time needed to manually derive termination for each benchmark. Development times for programs which can be completely specialized with an unsound termination criterion are dashed out. Missing entries represent benchmarks

on which the compared supercompiler failed to terminate after 30 minutes. All sampled numbers are provided at the 95% confidence level.

4.1 Interpretation

Our benchmarking results suggest there is potential to improve supercompilation using tailored termination criteria, which reduced program run times by up to 25% and an average of 8%. Total memory allocations were decreased by up to 100% and on average by 18.2% compared to vanilla GHC. We were able to successfully supercompile both microbenchmarks involving small functions, and more real-world programs like an implementation of RSA encryption.

There were certain examples where *chsc* took several minutes to finish. Additionally, several of the NoFib benchmarks (which were not designed with supercompilers in mind) did not finish within our 30 minute timeout. Most of these benchmarks contained functions with multiple recursive subcalls, suggesting that the tag-bag termination criteria does not signal a stop fast enough when there is an exponential path explosion. A minimal example that demonstrates the problem is quicksort with its two calls:

$$\begin{aligned} &qs\ (p : xs) = (qs\ lesser) +\!\!+ [p] +\!\!+ (qs\ greater) \\ &\quad \textbf{where}\quad lesser = filter\ (< p)\ xs \\ &\qquad\qquad\quad greater = filter\ (\geq p)\ xs \end{aligned}$$

On other benchmarks, including X2N1, Ackermann, SumSquare and EvenDouble, the tag-bag criteria terminated too early, before code paths with promising optimization opportunities were explored. Most of these benchmarks included mutually recursive functions.

The overhead for supercompiling programs was not very severe. On average, *chsc* increased compile times by 31,805%, while our supercompiler increased times by 105.8%. Presenting intermediate states in a graphical tree form that identifies where compilation starts to diverge makes user interactions straightforward, and introducing a user into the supercompilation loop only added an average of 1.04 minutes of active development time. Usually, it was immediately clear when a state should be flagged as terminal, as they lead to a pattern of repeated specialization without any meaningful changes.

In situations where supercompilation cannot perform any optimizations, performance was not degraded by code bloat. This is in contrast to previous supercompilers, which can degrade performance. The accumulator benchmark is an example of this phenomena: *chsch* produces a program which has 43% increase in the run time over the original program. All of our results are presented relative to GHC with optimizations enabled. If we disable these optimizations, we obtain up to 95% decreases in program run times, especially for the examples involving deforestation and fusion.

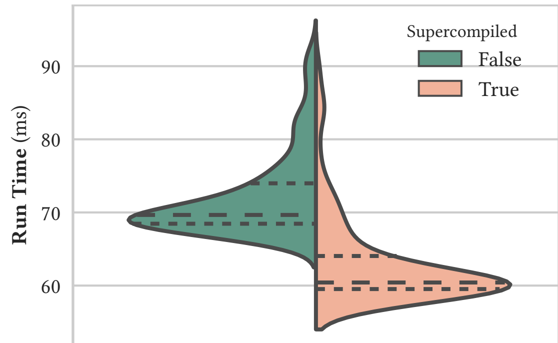

Figure 3. Violin plot of the EvenDouble2 across 10 million runs. The horizontal axis is a kernel density estimate and the vertical axis are the run times in milliseconds.

5 Related Work

Turchin first introduced the idea of supercompilation [22]. Ideas from supercompilation, including positive information propagation [19] and self-application [17] were integrated into languages like Refal in the 80s and 90s. These papers pushed more powerful supercompilation techniques but there were no practical implementations for more "mainstream" programming languages. There was a revival in interest in supercompilers in the early 2010s. Mitchell et al. applied these concepts to bring supercompilation to the Core Haskell language [16]. Mitchell was the first to use tag bags in termination criteria, as opposed to homeomorphic embeddings [14]. Jones and Bolingbroke further refined supercompilation in Haskell to support call-by-need based evaluation [3]. Bolingbroke also formalized a notion of sound general termination criteria using well-quasi-orderings [4]. Bolingbroke's work culminated in a PhD [5] thesis which collected techniques [2] for making call-by-need supercompilation tractable.

6 Future Work and Discussion

While relying on a user to specify when to terminate supercompilation is useful for evaluating the potential of custom termination criteria, we would obviously like to automate this process. One potential solution would be to adapt approaches to profile guided optimization [7] for conventional compilers, utilizing profiling information to find the optimal termination tags. Alternatively, as mentioned in section 4, some clear patterns emerge when visualizing the supercompilation process as a tree of specialization states. It may be possible to apply machine learning techniques to these trees in order to learn patterns which indicate overspecialization.

It may also be possible to create a language with annotations for dictating where supercompilation should cease, along the lines of the totality checker of *Idris* [6]. Consider such a theoretical language with the example from section 2.

Table 1. Results for benchmarks run with tailored termination and *chsc* compared with GHC in `-O2` mode.

	Program	Tailored Termination					Bolingbroke / *chsc*			
		[a] **Dev.**	[b] **Comp.**	[c] **Run**	[d] **Mem.**	[e] **Size**	[b] **Comp.**	[c] **Run**	[d] **Mem.**	[e] **Size**
NoFib	Bernouilli	1.0	+68.6% ± 2.1	-0.6% ± 7.1	0.00%	+20.5%	+154% ± 8.6	-0.3% ± 8.46	+6.58%	+31.5%
	Exp3_8	0.5	+9.6% ± 0.4	+0.5% ± 9.0	0.00%	+13.3%	+38.7% ± 1.7	+0.2% ± 6.3	+10.37%	+36.0%
	Regexps	2.5	+512% ± 63	-2.5% ± 4.2	+8.4%	+78.6%	–	–	–	–
	Integrate	5.0	+429% ± 18	-14.3% ± 4.1	-65.3%	+156.3%	+31805%	-4.1% ± 6.0	-67.6%	+151.1%
	Paraffins	1.0	+429% ± 44	-8.7% ± 4.7	-0.1%	+57.2%	–	–	–	–
	Primes	2.0	+224% ± 11	-6.3% ± 4.6	-15.9%	+68.0%	+21556%	-0.3% ± 8.6	-0.6%	+48.9%
	Queens	1.5	+55.7% ± 2.3	-9.4% ± 4.15	-27.4%	+73.0%	+715% ± 32	0.0% ± 8.4	+14.8%	+207.4%
	RFib	–	+2.7% ± 0.1	-0.7% ± 4.8	0.0%	9.9 %	+20243%	-0.1% ± 4.6	0.0%	+10.0%
	Tak	1.5	+49.4% ± 2.1	-3.1% ± 7.00	0.0%	+175.2%	–	–	–	–
	WheelSieve1	2.0	+124% ± 14	-9.7% ± 4.0	0.0%	+52.2%	–	–	–	–
	WheelSieve2	1.5	+681% ± 73	-8.3% ± 5.8	0.0%	+411.8%	–	–	–	–
	X2N1	0.5	+22.6% ± 1.4	-11.6% ± 6.3	-92.5%	+32.7%	+13.3% ± 0.7	-6.2% ± 5.6	-86.2%	+17.2%
	Coins	3.0	+113% ± 5	-12.4% ± 4.3	-38.4%	+146.3%	–	–	–	–
	RSA	5.0	+219% ± 17	-11.7% ± 4.4	-1.2%	+77.7%	–	–	–	–
Jonsson	Append	–	+2.1% ± 0.1	-1.8% ± 4.8	-12.5%	18.9%	+2.1% ± 0.1	+0.3% ± 5.2	-12.5%	19.0%
	Raytracer	–	+7.0% ± 0.4	0.0% ± 4.9	-51.7%	+15.7%	+7.1% ± 0.3	+0.3% ± 5.9	-51.7%	+15.6%
	SumTree	0.5	+38.6% ± 2.2	-14.4% ± 3.1	-11.1%	+38.6%	+18.2% ± 0.8	-7.3% ± 4.6	-19.9%	+16.1%
	SumTree2	0.5	+16.1% ± 1.8	-13.7% ± 2.6	-49.2%	+44.6%	+34.1% ± 2.5	+3.0% ± 10.9	-49.9%	+58.6%
	TreeFlip	0.5	+54.6% ± 1.8	-12.0% ± 6.6	-100.0%	+19.7%	+17.9% ± 0.8	-7.4% ± 8.4	-100.0%	+14.6%
	ZipMaps	–	+3.8% ± 0.3	-7.9% ± 4.8	-0.25%	-6.7%	+4.0% ± 1.1	-7.6% ± 6.7	-0.25%	-6.7%
	ZipTreeMaps	–	+5.0% ± 0.2	-17.0% ± 6.0	-2.4%	+0.8%	+5.2% ± 0.2	-15.3% ± 9.1	-0.2%	+1.2%
Bolingbroke	Accumulator	0.5	+5.5% ± 0.3	-9.5% ± 5.7	-0.2%	+21.5%	+329% ± 23	+43.3% ± 8.1	-13.4%	+36.7%
	Ackermann	1.0	+269% ± 17	-17.1% ± 9.4	-4.6%	+187%	+111% ± 5	-14.4% ± 6.7	0.0%	+13.0%
	Ackermann1	–	+1.9% ± 0.1	-4.5% ± 3.2	0.0%	-3.9%	+1.9% ± 0.1	-4.2% ± 4.1	0.0%	-3.9%
	Ackermann2	0.5	+4.3% ± 0.2	-4.1% ± 0.8	0.0%	+32.9%	+5.7% ± 0.3	-3.6% ± 0.6	-0.1%	+17.3%
	EvenDouble	1.5	+7.4% ± 0.4	-6.4% ± 4.6	0.0%	+114.9%	+6.2% ± 0.2	-4.6% ± 8.7	-0.2%	+35.8%
	EvenDouble2	–	+1.4% ± 0.1	-17.4% ± 7.6	-0.5%	-9.4%	+1.4% ± 0.1	-13.4% ± 6.5	-0.5%	-9.4%
	KMP	–	+12.0% ± 0.3	-9.0% ± 6.1	0.0%	+19.8%	+14.7% ± 0.7	-7.7% ± 4.7	0.0%	+50.5%
	LetRec	–	+0.9% ± 0.1	0.0% ± 8.4	0.0%	0.0%	+2.9% ± 0.2	+0.8% ± 7.1	0.0%	-6.2%
	MapMap	–	+1.1% ± 0.2	0.0% ± 4.5	0.0%	0.0%	+1.4% ± 0.1	+9.5% ± 7.2	-0.1%	+6.1%
	RevRev	1.0	+0.8% ± 0.1	-11.3% ± 4.2	-18.3%	0.0%	+8.2% ± 0.9	-11.8% ± 6.4	-18.3%	+87.3%
	SumSquare	0.5	+14.6% ± 2.0	-9.9% ± 3.2	-100.0%	+43.6%	+37.1% ± 2.0	+0.7% ± 7.7	-100.0%	+52.6%
	Min	0.0	+0.8%	-17.4%	-100%	-9.4%	+1.4%	-15.3%	-100%	-9.4%
	Max	5.0	+681.0%	+0.5%	+8.4%	+411%	+31805%	+43.3%	+14.8%	+207%
	Mean	1.04	+105.8%	-7.9%	-18.2%	+59.7%	+3005%	-2.1%	-19.5%	+36.2%

[a]Extra development time to resolve termination tags in minutes. [b]Change in compile time with supercompilation.
[c]Change in run time with supercompilation. [d]Percentage change in total bytes allocated with supercompilation.
[e]Change in program size with supercompilation.

The programmer annotates their code knowing that the specialized version should somehow involve a call to fold with xs as the third argument.

@terminateWhen(there is a call to $fold\ _\ _\ xs$)
$\lambda xs.fold\ (+)\ 0\ ([1, 2] \mathbin{++} xs)$

While this is a simple example, one can imagine a small domain specific language designed explicitly to guide the supercompiler for more complex functions.

7 Conclusion

Supercompilation is an automatic program specialization technique that can supersede many of the sophisticated optimization techniques that have been manually implemented in GHC [10]. Unfortunately, supercompilation has the potential to de-optimize programs by overspecializing them. Much work has been done to improve this state of affairs by improving termination criteria, including basing termination on tag-bags, and adding generalization and rollback, but the idea of fundamentally changing the termination criteria depending on the program being specialized is mostly unexplored. In this paper, we showed that tailored termination criteria can lead to significant performance improvements while avoiding the intricacies of general termination criteria. While our current manual approach for finding these criteria is clearly not scalable, we have demonstrated that optimization opportunities exist if we can automate the tailoring process.

Acknowledgments

We are thankful to all the friends and colleagues who reviewed this paper and for their many fruitful suggestions. We are also grateful to Max Bolingbroke for making their supercompiler open source and Simon Peyton Jones for their valuable discussion and insight.

References

[1] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F Sweeney. 2000. A comparative study of static and profile-based heuristics for inlining. In *ACM SIGPLAN Notices*, Vol. 35. ACM, 52–64.

[2] Maximilian Bolingbroke and Simon Peyton Jones. 2011. Improving supercompilation: tag-bags, rollback, speculation, normalisation, and generalisation. In *ICFP*. Citeseer, 2011.

[3] Maximilian Bolingbroke and Simon Peyton Jones. 2010. Supercompilation by evaluation. In *ACM Sigplan Notices*, Vol. 45. ACM, 135–146.

[4] Maximilian Bolingbroke, Simon Peyton Jones, and Dimitrios Vytiniotis. 2011. Termination combinators forever. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 23–34.

[5] Maximilian C Bolingbroke. 2013. *Call-by-need supercompilation.* Technical Report. University of Cambridge, Computer Laboratory.

[6] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593.

[7] Pohua P Chang, Scott A Mahlke, William Y Chen, and Wen-Mei W Hwu. 1992. Profile-guided automatic inline expansion for C programs. *Software: Practice and Experience* 22, 5 (1992), 349–369.

[8] Edgar C Fieller. 1954. Some problems in interval estimation. *Journal of the Royal Statistical Society. Series B (Methodological)* (1954), 175–185.

[9] WW Hisu and Pohua P Chang. 1989. Achieving high instruction cache performance with an optimizing compiler. In *Computer Architecture, 1989. The 16th Annual International Symposium on.* IEEE, 242–251.

[10] SL Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. 1993. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, Vol. 93.

[11] Peter A Jonsson. 2011. *Time-and size-efficient supercompilation.* Ph.D. Dissertation. Luleå tekniska universitet.

[12] Peter A Jonsson and Johan Nordlander. 2009. Positive supercompilation for a higher order call-by-value language. In *ACM SIGPLAN Notices*, Vol. 44. ACM, 277–288.

[13] Tomas Kalibera and Richard Jones. 2013. Rigorous benchmarking in reasonable time. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 63–74.

[14] Michael Leuschel. 1998. On the power of homeomorphic embedding for online termination. In *International Static Analysis Symposium.* Springer, 230–245.

[15] Neil Mitchell. 2010. Rethinking supercompilation. In *ACM Sigplan Notices*, Vol. 45. ACM, 309–320.

[16] Neil Mitchell and Colin Runciman. 2007. A supercompiler for core Haskell. In *Symposium on Implementation and Application of Functional Languages*. Springer, 147–164.

[17] Andrei P Nemytykh, Victoria A Pinchuk, and Valentin F Turchin. 1996. A self-applicable supercompiler. In *Partial Evaluation.* Springer, 322–337.

[18] Will Partain. 1993. The nofib benchmark suite of Haskell programs. In *Functional Programming, Glasgow 1992.* Springer, 195–202.

[19] Morten Heine Soerensen, Robert Glück, and Neil D. Jones. 1996. A positive supercompiler. *Journal of Functional Programming* 6, 6 (1996), 811–838.

[20] Morten Heine Sørensen, Robert Glück, and Neil D Jones. 1994. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In *European Symposium on Programming*. Springer, 485–500.

[21] Michael Sperber. 1996. Self-applicable online partial evaluation. In *Partial Evaluation.* Springer, 465–480.

[22] Valentin F Turchin. 1986. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8, 3 (1986), 292–325.