The TIL/ML Compiler: Performance and Safety through Types

G. Morrisett D. Tarditi P. Cheng C. Stone R. Harper P. Lee
jgm@cs.cornell.edu dtarditi|pscheng|cstone|rwh|petel@cs.cmu.edu

January 28, 1996

1 The Problem

Systems code requires both high performance and reliability. Usually, these two goals are at odds with each other. For example, to prevent kernel data structures from being over-written or read, either accidentally or maliciously, conventional systems use hardware-enforced protection or software fault isolation (SFI) [18]. Unfortunately, both of these techniques exact a cost at run time: Hardware protection requires expensive context switches and data copying to communicate with the kernel or other processes, whereas SFI requires run-time checks for loads, stores and jumps. Furthermore, these protection mechanisms only guarantee "read/write" safety and are often too weak to guarantee the integrity of a system. For instance, systems code must still verify at run time that a capability (i.e., a file descriptor, port, message queue, *etc.*) is valid when using SFI or hardware protection.

In contrast, systems designed around strong, statically-typed languages such as Modula-3 and Standard ML (SML), provide abstraction mechanisms, including objects, abstract datatypes, polymorphism, and automatic memory management, that enforce read/write safety as well as capabilities at *compile-time*. For instance, we can define an abstract type `message_queue` with appropriate operations using the `abstype` mechanism of SML. The type system of SML prevents programmers from forging `message_queue` values either accidentally or intentionally. As a result, we need not check at run time whether or not a `message_queue` argument is valid. Thus, in principle, using static type checking should provide superior performance to either hardware protection or SFI, simply because checks are performed at compile or link time instead of run time.

Unfortunately, two impediments have kept the "programming language" approach to systems software from becoming a reality. The first problem is that implementations of languages such as Modula-3 and SML restrict representations of values to support features like polymorphism and garbage collection. In particular, implementations tag values to support garbage collection and run-time type dispatch (e.g., Modula-3's `typecase` or SML's polymorphic equality), and box objects (i.e., force them into a machine word) to support polymorphism and abstract data types.

Systems code requires direct access to underlying hardware or subsystem data structures, and thus cannot afford to tag or box objects. For instance, it is often necessary to read/write specific 32-bit (or 64-bit) integer values into device registers, frame buffers, DMA regions, or page tables. As a result, implementations that tag integers by stealing a bit cannot directly access such devices. Without direct access to the hardware or data structures, the performance benefits of static type checking become irrelevant.

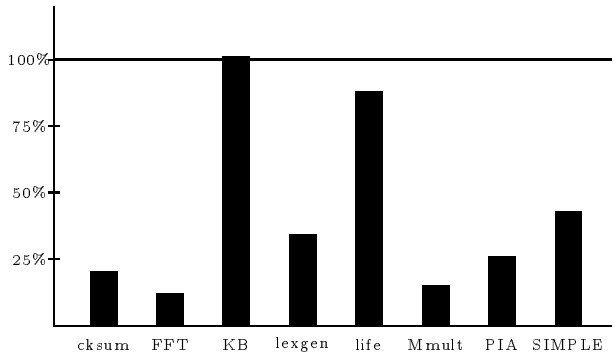

Figure 1: TIL Execution Time Relative to SML/NJ

The second problem is that, whereas languages like SML provide a rich type system that eliminates more run-time checks than for example Modula-3, implementations of SML-like languages produce code that performs poorly, especially when compared to imperative languages like C. For instance, we have observed up to a 10-fold increase in running times for small benchmark programs compiled under Standard ML of New Jersey (SML/NJ) when compared to C. We argue that part of the performance problem of SML/NJ code is due to the need to tag and box objects to support polymorphism and garbage collection. But another large part is due to implementors concentrating on making function calls fast, instead of concentrating on generating good code for loops and recursive functions.

2 The Solution: TIL

Having identified these key implementation problems, we have constructed a new compiler for SML called TIL with the following desirable properties:

1. There are almost no restrictions on data representations. In particular, we support "nearly" tag-free garbage collection and box-free polymorphism.

2. Code produced by TIL is efficient: On a sample set of benchmarks, TIL code is roughly 3 times faster and allocates one-fifth the data compared to SML/NJ version 108.3, running on a DEC Alpha 3000/300 LX. (See Figures 1 and 2 and Tables 2 and 3)

3. All but the last of the intermediate languages used by TIL are strongly typed.

We briefly discuss each of these points below. Readers interested in more details regarding the internals of TIL and its performance should see [16, 11, 15].

To eliminate restrictions on object representations, TIL uses a combination of *intensional polymorphism* [7] and *"nearly" tag-free garbage collection* [12]. Intensional polymorphism allows programs to perform dynamic type analysis, much like the `typecase` construct of Modula-3. However, intensional polymorphism does not tag values with their types. Instead, representations of types are passed as independent arguments to polymorphic functions so that values can have natural representations. Since values are not tagged, type arguments among multiple values can often be shared. For instance, a function that takes an array of values of uniform, but unknown type requires only one type argument.

Program	lines	Description
Checksum	241	Checksum fragment from the Foxnet [2], doing 5000 checksums on a 4096-byte array.
FFT	246	Fast fourier transform, multiplying polynomials up to degree 1,000,000.
Knuth-Bendix	618	An implementation of the Knuth-Bendix completion algorithm.
Lexgen	1123	A lexical-analyzer generator [1], processing the lexical description of Standard ML.
Life	146	The game of Life implemented using lists [13].
Matmult	62	Integer matrix multiply, on 100x100 integer arrays.
PIA	2065	The Perspective Inversion Algorithm [19] deciding the location of an object in a perspective video image.
Simple	870	A spherical fluid-dynamics program [4], run for 4 iterations with grid size of 100.

Table 1: Benchmark Programs

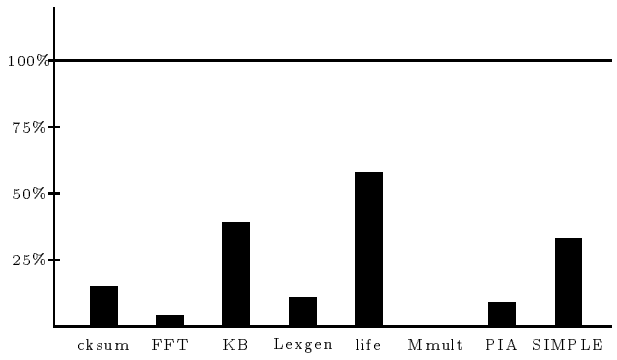

Figure 2: TIL Heap Allocation Relative to SML/NJ

Our approach to tag-free garbage collection is to keep integers, floating point numbers, other values that fit into registers (e.g., bools), and values allocated on the stack tag-free. Instead, we associate type information with code points at compile time and use this information during garbage collection to locate live, traceable values that reside in either registers or on the stack. In particular, we use the return addresses in stack frames to locate type information so that we can determine which registers and which stack slots contain pointers to heap-allocated values. We use the type arguments of functions to determine whether or not values of unknown type are pointers. Values allocated on the heap are tagged with a header word (or words) describing which components of the value are pointers. This approach is similar to languages like C and C++ where heap-allocated data structures (i.e., `malloc`ed values) are tagged with header words. However, unlike so-called "conservative" collectors (e.g., [3]), our approach always collects unreachable objects. Note that our approach does not preclude mapping tag-free values outside the heap (such as a page table or frame buffer) into our address space. Furthermore, our approach easily extends to fully tag-free collection in the style of Tolmach [17].

We use a combination of standard functional language optimizations such as inlining, uncurrying, constant typecase-elimination to eliminate polymorphic and higher-order functions without blowing up code size. We also apply conventional loop optimizations, including common subexpression elimination and loop-invariant removal, to recursive functions. These loop optimizations are quite important and provide roughly a factor of 2 speed improvement for most benchmarks.

Because the intermediate forms of TIL are strongly typed, we can verify the type safety of the output of all but the very last phases of the compiler. This has helped to identify and eliminate various bugs in the compiler itself. But having strongly-typed intermediate forms has an additional benefit: Since the output of the optimizer is type-safe, and because the bulk of compile time in TIL is spent in optimization, we can ship this optimized code to untrusting sites (e.g., a web-browser or kernel). These sites can verify the type safety of the low-level, optimized code and then locally perform only the last few stages of compilation.

Using typed intermediate languages forced us to push the frontier of type systems in order to support static checking of *lower-level* C-like languages instead of *higher-level* SML-like languages. The key difficulty is that much of the type safety in a language like SML is provided by the high-level constructs, including closures and abstract datatypes. We must expose the underlying representations of these constructs to an optimizer in order to get good code. But as we expose the representations, we need more powerful type systems to track the invariants that were guaranteed at the source level.

For example, an important phase in TIL eliminates closures (higher-order functions) by representing them as records consisting of code and an environment. After this phase, the optimizer can eliminate redundant environment operations or redundant closure constructions. These optimizations are not possible before the representations of closures are made explicit. But exposing the representations of closures requires additional support from the type system. To ensure type safety, we must guarantee that only the environment of a given closure is passed to the code of that closure.

We addressed this closure issue by using a combination of *existential* types [10] together with *translucent* types [6, 8] to support safe, explicit representations of closures [9]. Simply-typed intermediate forms, like the ones used in the Java Virtual Machine [5] and the SML/NJ compiler [14], cannot ensure these properties at compile time. As a result, these systems must sacrifice either safety or performance.

3 Summary and Conclusions

We view the application of advanced type theory in compilers as one of the most promising means for achieving both performance and safety in systems code. The TIL compiler shows that languages like SML can provide safety through types without restricting data representations. Furthermore, TIL demonstrates that striking improvements in SML code performance can be achieved through the combination of intensional polymorphism, nearly tag-free garbage collection, and fairly conventional optimizations. We have found that much of the needed type theory is available "off the shelf", but compiler writers have yet to take advantage of it. Conversely, much of the research in type theory is oriented towards high-level languages, whereas the application of types to low-level intermediate languages seems to hold the most promise.

References

[1] A. W. Appel, J. S. Mattson, and D. Tarditi. A lexical analyzer generator for Standard ML. Distributed with Standard ML of New Jersey, 1989.

[2] E. Biagioni, R. Harper, P. Lee, and B. Milnes. Signatures for a network protocol stack: A systems application of Standard ML. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 55–64, Orlando, Florida, June 1994. ACM.

[3] H.-J. Boehm. Space-efficient conservative garbage collection. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 197–206, Albuquerque, June 1993.

[4] K. Ekanadham and Arvind. SIMPLE: An exercise in future scientific programming. Technical Report Computation Structures Group Memo 273, MIT, Cambridge, MA, July 1987. Simultaneously published as IBM/T. J. Watson Research Center Research Report 12686, Yorktown Heights, NY.

[5] J. Gosling. Java intermediate bytecodes. In *ACM SIGPLAN Workshop on Intermediate Representations (IR'95)*, Jan. 1995.

[6] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the 21st Annual ACM Symposium on Principles of Programming Languages*, pages 123–137, Portland, OR, Jan. 1994. ACM.

[7] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 130–141. ACM, Jan. 1995.

[8] X. Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the 21st Annual ACM Symposium on Principles of Programming Languages*, pages 109–122. ACM, Jan. 1994.

[9] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Programming Languages*. ACM, Jan. 1996. To appear.

[10] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Trans. Prog. Lang. Syst.*, 10(3), 1988.

[11] G. Morrisett. *Compiling with Types*. PhD thesis, School of Computer Science, Carnegie Mellon University, Dec. 1995. Technical Report No. CMU-CS-95-226.

[12] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *ACM Conference on Functional Programming and Computer Architecture*, pages 66–77, La Jolla, June 1995.

[13] C. Reade. *Elements of Functional Programming*. Addison-Wesley, Reading, Massachusetts, 1989.

[14] Z. Shao and A. W. Appel. A type-based compiler for Standard ML. In *Proceedings of the ACM SIGPLAN '9Language Design and Implementation*, pages 116–129, La Jolla, California, June 1994. ACM.

[15] D. Tarditi. *Optimizing ML*. PhD thesis, School of Computer Science, Carnegie Mellon University. In preparation.

[16] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ml. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, May 1996. To appear.

[17] A. Tolmach. Tag-free garbage collection using explicit type parameters. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 1–11, Orlando, FL, June 1994.

[18] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *14th ACM Symposium on Operating Systems Principles*, Dec. 1993.

[19] K. G. Waugh, P. McAndrew, and G. Michaelson. Parallel implementations from function prototypes: a case study. Technical Report Computer Science 90/4, Heriot-Watt University, Edinburgh, Aug. 1990.

Program	Exec. time (s)		TIL/NJ
	TIL	NJ	
Checksum	11.81	57.89	0.20
FFT	5.45	43.97	0.12
Knuth-Bendix	7.47	7.42	1.01
Lexgen	3.50	10.32	0.34
Life	0.67	0.76	0.88
Matmult	0.48	3.22	0.15
PIA	1.23	4.79	0.26
SIMPLE	37.39	86.07	0.43
Geo. mean			0.32

Table 2: Comparison of running times

Program	Heap alloc. (Kbytes)		TIL/NJ
	TIL	NJ	
Checksum	143,898	984,775	0.15
FFT	9,107	214,855	0.04
Knuth-Bendix	36,942	94,493	0.39
Lexgen	11,919	111,252	0.11
Life	3,563	6,096	0.58
Matmult	0	30,990	-
PIA	4,089	53,850	0.09
SIMPLE	265,212	803,119	0.33
Geo. mean (excluding Matmult)			0.17

Table 3: Comparison of heap allocation