Compiling Erlang via C

Geoff Wong
Software Engineering Research Centre,
110 Victoria St, Carlton 3053, Australia.
email: geoff@serc.rmit.edu.au
phone: +61 3 9925 4011, fax: +61 3 9925 4094

Abstract

This paper discusses the implementation of an Erlang compiler which provides native compilation using C as an intermediate language. The intermediate output, while not elegant, is amenable to being ported away from the Erlang environment and into more 'traditional' C based development environments.

The compiler also provides a number of special features. These features include support for a concurrent monitoring system and simple static type analysis. Each Erlang module is compiled as a native shared object library to support dynamic code loading and unloading. In contrast to the standard Erlang implementation the runtime system is tightly integrated into the Unix environment. Erlang processes are instantiated as actual Unix processes at runtime. This allows Unix system analysis and monitoring tools to be used upon individual Erlang processes.

Overall this compiler offers linear some computational performance improvements over standard Erlang systems while being highly portable and producing a re-usable intermediate code.

Keywords Erlang, C, compilers, concurrency, performance.

1 An Erlang Compiler

The Erlang compiler, *Gerl*, described in this paper is a tool implemented as part of the support framework for a continuous software monitoring system. The compiler is intended to instrument system application code to automatically provide this monitoring system for the developer. The monitoring framework is described in [14].

Proceedings of the 1998 Computer Science Postgraduate Students Conference, Royal Melbourne Institute of Technology, Melbourne, Australia, December 8, 1998.

There were a number of motivations for developing a new compiler. Primarily it was important that the design of the compiler included monitoring support as a central design feature of the system. It also allows the monitoring framework support to be built into the compiler in a modular way. This is important as the framework support is intended to be reused within other compilers. And finally it enabled the development of a fuller understanding of the mechanisms and performance issues related to implementing the monitoring framework.

In developing the compiler a number of interesting implementation issues were explored:

- Translation of a functional language into an imperative form, particularly function polymorphism, pattern matching, last call optimisation and exception handling.
- Implementing Erlang's highly concurrent process model.
- The issue of efficient garbage collection.
- The runtime support environment including integration with Unix and the runtime loading, linking and updating of compiled code.
- The performance of the compiler versus the existing virtual machine based implementations.

This following sections discuss each of these issues in more detail. Problems areas and implementation choices are outlined and the chosen solutions are presented.

2 Translating Erlang to C

Erlang code is compiled using C as an intermediate language. C offers a variety of features that make it useful as a low-level target language, compilation to C has become an increasingly popular choice for language implementors [6]. Despite this there are significant obstacles to overcome when compiling Erlang into C. These obstacles relate directly to the differences between the languages.

Erlang is designed to provide a concurrent functional programming language which is targeted at real-time concurrent systems such as communications switches. As a (mostly) functional language Erlang provides a high degree of polymorphism, dynamic type checking, resizable lists, tuples, pattern matching, inter-process communication, exception handling, incremental (real-time) garbage collection, a code visibility mechanism and reliance on last call optimisation to provide bounded space iteration for tail-call recursion.

Contrast this to C which provides a statically typed language with a flat namespace for function names, immediate variable assignment, explicit memory allocation, no support for polymorphic function calling, no language support for inter-process communication, no direct language support for resizable objects, stack frames of differing organisations on different architectures, no direct access to the stack frames, and no support for `goto` outside a function (or unknown effects if one does with **GNU gcc** [4]). C provides a large library of support code, with functions which link directly into the operating system. C is also one of the most widespread and mature languages and C compilers are available for most computer architectures.

While C provides many benefits as an intermediate language [6], translating Erlang into C still provides a significant challenge. *'The trouble is that C was designed as a* programming *language not as a compiler target language'* [10]. A better target would be the proposed language C−− [10], unfortunately this language was not at a stage of development that made it useful for the construction of this compiler.

Translating Erlang into code suitable for **GNU gcc** [4] automatically provides portability to all architectures supported by the widely available **GNU gcc** compiler. By using **GNU gcc**'s version of C there is a clear emphasis on portability since **GNU gcc** is a very widely ported compiler. This is traded off against slower compilation speeds and possibly slower object code (versus native compilation). Slower compilation speed impacts upon one of Erlang's more effective uses which is that of a rapid prototyping language. This impact is mitigated by the continuing existence of interpreted versions of Erlang.

Using **GNU gcc** also provides access to excellent native code generation, ability to use C-level debuggers and access to a mature compiler which provides high quality optimisation, dead code removal and numerous other features that normally take significant time to develop. The libraries associated with **GNU gcc** provide a convenient interface to existing system code. A sample Erlang module is shown in Figure 2, the corresponding C code that is produced as the intermediate compilation step is shown in Figure 3.

With a little effort different C compilers, which conform closely to the ANSI standard, can be used as the intermediate step compilers. Compiling using C as an intermediate language typically provides performance well beyond that of interpreted system and comparable to many native code compilers. This is clearly demonstrated in the results of the widely applied Pseudoknot Functional Benchmark [5].

A future goal, given the unavailability of a general purpose portable assembly language such as C−−, is to compile into a form appropriate for the backend of **GNU gcc** [4]. This would skip the (slow) intermediate C step while still providing us access to most of the aforementioned advantages and the possibility of allowing direct support for pre-built tools such as the **GNU gdb** debugger [12].

2.1 Type Checking

To support the dynamic type checking that is required by Erlang, values tagged with types are used throughout compilation and in the runtime system. Runtime values (or objects) are effectively bundles of data with an associated type tag; these types can be passed to functions with matching parameter sequences. The internal description of an Erlang value, used throughout the C code, can be seen in Figure 1.

The compiler deviates slightly from the defined Erlang standard [1] by providing limited type-checking where possible. The type checking is limited by the non-typed nature of Erlang; extra syntactic support for type specification within the language maybe an area worth further exploration. A number of proposals for type systems that extend Erlang already exist [11]. These type systems seek to increase early error detection without impacting Erlang's existing code base and Erlangs use as a rapid prototyping language.

Type checking and inferencing can provide us with the ability to optimise out type-tags in certain instances, particularly for integers and floats, within a function. This can provide a dramatic increase in performance. Type checks are implied by some pattern matching and can be added directly as C `if` statements to ensure the code after the statement doesn't need to check type tags on values.

File: erval.h

```
typedef union
{
    int number;
    float real;
    char * atom;
    char * string;
    char * var;
    struct _tuple * tuple;
    struct _list * list;
    struct _binary * binary;
    void * any;
} Many;

typedef struct value
{
    Etype tag;
    Many u;
} Val;
```

Figure 1: Basic Internal Value Structure

2.2 Pattern Matching

Pattern matching is one of the key features of functional languages which distinguishes them from imperative languages. A pattern maybe given for function clause parameter lists, case clauses, `if` clauses, `receive` clauses and in the equivalent of an imperative assignment. Figure 2 has example pattern matches for lists, with or without contents, in the parameter lists of the function clauses. Emitting code that assigns values to variables created in these pattern matches is handled in stages:

- Initially an abstract syntax tree (AST) is created in the parser.
- Symbols in the AST are then added into the appropriate symbol table.
- Pattern matching code is emitted by tree traversal of the pattern matching code.
- During the traversal the 'route' is passed down the tree to the leaves (variable nodes) and an appropriate assignment is constructed.

Function clause pattern matching is implemented by combining a group of polymorphic Erlang function clauses, accepting the same number of parameters, into a single C function. This is implemented as a series of `if` statements for each possible case of the pattern matches. Pattern matching at runtime is then performed internally by executing these `if` statements to check for the correct pattern formations before variables are assigned. The `when` guards associated with

File: reverse.erl

```
-module(reverse).
-export([reverse/1]).

reverse(L) ->
        reverse(L, []).

reverse([H|T], L) ->
        reverse(T, [H|L]);

reverse([], L) ->
        L.
```

Figure 2: Erlang Code: 'Reverse List'

function clauses are implemented as `if` statements inside a function clauses `if` pattern match. If the guard is not matched then the flow of control falls through to check the next function clause pattern match.

As the code generated is a series of if/else statements which are relatively slow to execute, further static analysis maybe able to provide pattern matching improvements.

2.3 Module Naming and Linking

The *module:function/parameter* combination of Erlang functions provides a unique name within the Erlang runtime name space. An equivalent naming scheme must be supported for functions translated into C. It is important that this scheme not be restricted to individual compilations because modules can be dynamically linked and loaded at runtime at arbitrary times in the future. Dynamic linking and loading is directly supported by the C dynamic loading libraries available in modern versions of Unix.

Function polymorphism is managed by translating identically named functions with the same number of parameters into a single corresponding C function, see section 2.2. This allows each C function to have the number of parameters appended to the name of the C function. This distinguishes the function from other functions in the module with the same name. Functions are uniquely distinguished from each other across modules by having each function prepended with the name of the module. This provides satisfactory and simple scheme for a limited number of modules.

The basic naming scheme is further complicated if support is provided for the standard Erlang module replacement scheme. This scheme allows two versions of a module to exist at once. To avoid

File: reverse.c

```
#include "erlib.h"
Val reverse_reverse_1(Val p0);
Val reverse_reverse_2(Val p0,Val p1);
Val reverse_run_1(Val p0);
_Symbol * reverse__exports()
{
        static _Symbol tmp[] = {{"reverse_1",1},
        {"reverse_2",2},
        {"run_1",1},
        { NULL, 0 } };
        return tmp;
}
Val reverse_reverse_1(Val p0)
{
Lreverse_reverse_1:
        if ((1))
        {
        Val L = p0;
        Val _ret, _vx0, _vx1;
        _ret=EValcopy((_vx0 = L,_vx1 =
(Emake_list(0)),
         reverse_reverse_2(_vx0,_vx1)));
        return _ret;
        }
        Ethrowv("Failed to match function
clause reverse/1.\n\t%s\n",ESV(p0));
}
Val reverse_reverse_2(Val p0,Val p1)
{
        Val __vvv;
        int recur = 0;
        void * sp = set_stack;
Lreverse_reverse_2:
        if ((Eis_full_list(p0)) && ((1)) && (1))
        {
        Val H = (p0.u.list→e);
        Val L = p1;
        Val T =
(Etail_list(__vvv,p0,0,"reverse:reverse/2"));
        Val _ret, _tx1, _tx2;
        p0 = T;
        p1 = (_tx2=L,Econcatvl(_tx1,(H),_tx2));
        cut_stack(sp);
        recur = 1;
        goto Lreverse_reverse_2;
        }
        if ((Eis_empty_list(p0)) && (1))
        {
        Val L = p1;
        Val _ret;
        _ret=EValcopy(L);
        return _ret;
        }
        Ethrowv("Failed to match function
clause
reverse/2.\n\t%s\n\t%s\n",ESV(p0),ESV(p1));
}
```

Figure 3: C Translation of 'Reverse List'

name clashes, version information must be included as part of the module name (at the C level). Function calls internal to a module are translated directly to these names. External calls, which are wrapped in a library function call, need not include this information. The internal library calling mechanism resolve calls to call the correct version of the function required. This is implemented by the maintenance of a table of modules; a lookup of this table is performed and the current version is extracted. This scheme is easily extended to allow for the maintenance of any number of versions of a module in memory.

For very large development projects the flat name space provided for Erlang modules may prove inadequate and some form of hierarchical naming would need to be supported. A hierarchal naming scheme would also allow different modules with the same basic module name to be used together in a single system.

2.4 Last Call Optimisation

Tail recursive functions are ones which allow the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure [3]. Essentially a tail call is one which can be evaluated without maintaining the previous function call's stack frame upon the stack. The return value does not rely upon any data within that stack frame. *Last call optimisation* (LCO) allows the compiler optimise such functions so that the stack uses bounded space. LCO is used widely in Erlang because it allows processes to exist for a significant amount of time, often as *servers*, without exhausting the memory of a system.

Translating functions that maybe last call optimised into C provides significant challenges. There are two stages to this process: detecting functions that can be optimised in this way and then implementing code which does the optimisation. Numerous approaches have been taken to implementing LCO in C, these include [10]: embedding entire programs in a single C function, using parameterless C procedures which return the address of the next function to call, post-processing assembly code to do the optimisation, and using non-ANSI extensions to transfer control outside a function.

Some of these solutions require extreme reorganisation of 'straightforward' code emission techniques. The compiler implements LCO within self-recursive functions by using `goto` and passing the parameters on the heap (instead of the local stack frame). Parameters are freed on subsequent recursions.

If the LCO is local to a function then local variables need to be reassigned with the new calling values and a `goto` executed after the previous stack frame is rewound. This keeps the machine stack space used limited to its initial allocation. A difficulty with this process is the extensive use of `alloca()` within the compiler. `Alloca` allocates stack memory which is not released by a `goto`. De-allocation is done by implementing (in assembly) a function pair; one which returns the current stack points (into a variable) and the other which can retract the stack to that location. Alternately this stack retraction can be managed using a `setjmp()`, `longjmp()` pair instead of `goto` and the pair of assembly instructions.

The compiler uses the technique described in [6] to implement LCO outside single functions. Last call optimisation between functions is implemented by adding a small amount of assembly 'glue' into each function. This assembly 'glue' simply declares a global label, within the function, which can be jumped to directly using `goto`. Providing the number of parameters of the called function is strictly less than or equal to the calling function. Is is possible to wrap this assembly 'glue' in a series macros as the Mercury compiler [6] does. This increases the portability of the compiler by allowing it to be ported to other system without the assembly functions being implemented.

2.5 Exception Handling

Erlang exception handling is based upon the use of the language mechanism `catch()` and the built in function `bif:throw/0`. Exceptions are generally handled within a process unless that process is *linked* to another process. A process propagates any unhandled errors (causing it to fail) to all linked processes.

Expression based exception handling is fundamentally difficult within C. The use of `setjmp()` and `longjmp()`, which are often used to implement `catch()` (and `throw()`), requires the support of an exception stack and as such is relatively slow and can cause other difficulties particularly relating to garbage collection. This occurs because execution jumps immediately to the location of the last `setjmp()`. Within our system, heap allocated values requiring explicit de-allocation by statements embedded into the code are not executed. Although unlikely, this has the potential to cause serious space leakage problems in recursive functions which repetitively have the same error occurring.

The mechanism for handling exceptions with the framework is important as the monitoring framework must have the capability to catch exceptions and also propagate them onto monitoring processes. The current Erlang system supports this linkage for unhandled errors by attaching the monitoring process to the functional process. A mechanism is required to forward handled exceptions to linked processes. This mechanism is not directly provided by the C language as standard, the addition of a small amount of runtime support code is required to implement this mechanism.

3 Memory Management

3.1 Memory Allocation

Erlang automatically reclaims data objects when they are no longer in use. The memory management scheme is hidden from the programmer. Because of the functional nature of Erlang code objects created within an Erlang process have limited persistence. Objects are maintained in two places: within the process's runtime activation record or within the processes incoming message queue. When a process is complete its data is destroyed. As a result of this activation record based existence it is possible to build a primarily stack based object allocation scheme. Basic values such as integers and floats are easily copied to/from the stack in every operation and are ideally stack allocated.

The initial implementation of the compiler uses a blend of a stack based and heap based allocation scheme. All values, except return values and list elements, are allocated upon the stack so they are automatically garbage collected when the function returns. Because it is not possible to know the size of a return value, and hence it is not possible to allocate the space for the return value in the calling activation record, return values are allocated on the heap. Given a specialised intermediate compilation (ie. not C) language this maybe possible.

Atoms are initially stack allocated unless they are returned from a function call to a stack frame 'above' which they were allocated in. The stack can be used as temporary storage for atoms. Atoms of this nature may also be better supported by a *shared string* (in this case shared atom) support mechanism.

More complicated structures such as tuples and lists requiring significantly more work to handle within a stack based allocation scheme. Tuples can be handled in the same manner as atoms. This can be done efficiently if each tuple is managed as single blocks of memory (and copied by `memcpy()`).

'Deep' copying is not required providing all the elements of a heap allocated tuple are also heap allocated.

Lists are more difficult to handle because they can be self-referential which makes them difficult to block copy. One of the most common list operations is appending to the head of a list which makes also makes a block allocation and copying scheme for lists expensive. As a result list elements are always heap allocated. The head of the list is treated as a 'normal' value.

3.2 Garbage Collection

The Erlang language standard [1] does not specify a particular type of garbage collection, leaving it up to the implementor to select a scheme. Nevertheless the garbage collection scheme used has a substantial impact upon the overall and perceived performance of a system. Because the compiler translates into C as an intermediate language it is difficult to use an accurate garbage collector because C pointers contain no excess information. This problem is alleviated as a result of Erlang's dynamic typing system. Every type is associated with a type tag. These type tags can provide extra space to store information required by the garbage collector.

A non-incremental mark and sweep [13] garbage collection scheme can result in processes being halted for a (relatively) significant period of time, but these stoppages are perceived as significant for users involved in interactive applications. Because of Erlang's concurrent nature this type of impact is less significant (than the typical mark and sweep which halts an entire system), in this case only a particular process would be halted. Nevertheless artifacts such as this are undesirable for real-time systems.

Internally the type of garbage collection has a substantial impact on how data types are stored and collected and as a result how errors are handled. For example a reference counting garbage collection scheme is defeated by a `longjmp()` implementation for `bif:throw/0`. Cleanup code at the end of each block is not executed because the activation stack is not unwound in the normal manner. Hence reference counts are not decremented. A separate allocation stack would need to be maintained and each `longjmp()` would need to be associated with an effort to decrement reference counts on allocated objects upon this stack.

By removing reliance on heap based allocation memory can be reclaimed 'for free' when a function is completed. This is effective for all values except those that use the heap as a return value space and for list elements. Return values which are not required are cleared explicitly in the calling code after they are used. This presents some garbage collection problems in the event of an exception with garbage collecting code not being executed.

This method of memory reclamation is the 'standard' way of reclaiming temporary variables in a block structured language. It works effectively in Erlang for a number of reasons:

- Values are passed by value (hence copied) when sent to another process.
- Values have no global scope to exist in and become inaccessible once a function is completed.

This implementation of memory allocation attempts to be stack based where it is efficient and uses the heap as an infinitely sized return register. Because return values must be copied back into the local stack space. Return values often consist of large lists and tuples in Erlang.

The drawback of a stack based allocation with individual processes for Erlang processes is that data structures must be copied to be passed around and data sharing optimisations cannot be made. This can be a significant performance penalty if the system passes a lot of large structures within messages between processes.

4 Concurrency

A primary feature of Erlang is its management of multiple processes as a fundamental part of the system. This feature is critical to the construction of a monitoring Erlang compiler as it allows us to implement the dual process monitoring system using language mechanisms that must be made available. This monitoring processes are provided by using the compiler to include code in each functional process to automatically spawn a monitoring process.

Concurrency in Erlang relies on two language features and a number of standard functions. The `bif:spawn/4` function creates a new process and immediately returns the process identifier to the spawning process. All communication in between processes is done via a message passing system. Any process may communicate with another process providing it has its process identifier; this arrangement also doubles as a basic security system in Erlang. Messages are sent with the '!' primitive [7].

Each process has a message queue. Messages accumulate in the receiving processes message queue in the order they arrive. A process retrieves messages from the message queue using the `receive` primitive and a number of associated pattern matches. Only messages which match one of these patterns (for a particular receive) are processed. If a message doesn't match it is left on the message queue until another receive is invoked and the entire message queue for that process is checked again for a match. Without a 'catchall' clause this can lead to overflow problems in message queues.

Although eventually intended for a multiple processor environment where direct control of the use of processors is available the existing system is built upon multiple processor Unix machine. Since the multi-processing support generally places processes on separate processes where possible we can derive reasonable performance tests of the compiler by running repeated tests upon the machine in different configurations (with different test packages).

4.1 Erlang Nodes

An *Erlang node* is a group of processes which share the same process name table. This name table is handled as a separate process which allows a process to look up another process by a registered name. Typically there is one of these nodes per machine but it is possible to have multiples nodes per machine. A monitoring process belongs to the same node its dual functional process.

Because of the non-portability of asynchronous message handling techniques between different operating systems all messages are sent via this central process. This restricts machine dependent code to this process and it allows it to use a message polling method, rather than an interrupt driven method, to gather incoming messages. This is advantageous because portability of message polling code is significantly higher than for interrupt driven message collection. The process is then responsible for signalling each process when a message arrives (using Unix signals).

An Erlang node also acts as a central message collection repository on each machine. It waits and receives messages from any process attempting to communicate with processes that are part of that node. On message reception the intended receiving process is signalled by the central repository (Erlang node). That process attempts to retrieve that message next time it executes a `receive` instruction.

Currently this is implemented using a simple communication protocol which requires individual receiving processes to request the incoming message from the central message repository. If a request message does not match one of the patterns in the `receive` clause that message is queued locally within the process and another `receive` attempt is made. Each process also has its own message queue for messages which are collected from the central repository, this queue is checked when a different `receive` in that process is executed.

4.2 Interprocess Communication

Each process has a message queue associated with it. Subject to an implementation specific limit, all messages to an object build up in this queue. When a process executes a `receive` statement it retries a message matching a given pattern from the message queue.

Process communication is implemented using internet domain (TCP/IP) Unix sockets. This allows Erlang processes to be spread across a number of machines as well as across multiple processors on the local host. This support is provided by linking each compiled module with an Erlang runtime library which provides communication support and other Erlang specific language support mechanisms. Processes communicate via sockets with a combination of the machine address and their Unix process identifier as the socket identifier. This identifier is created when a processes is spawned and can be maintained by other processes wishing to communicate with the spawned process.

The process identification numbers of each process have their location encoded in them. This enables them to be shared among processes and still have the spawned process being contacted successfully by another process irrespective of its location. Process identifiers must include the machine IP number so as to be unique when passed between multiple machines, the internal process identifier format that is currently used:

$$< \text{IP\#, Port, Unix PID} >$$

System performance depends greatly upon efficient message passing in a concurrent environment such as the one Erlang provides. When passing messages between machines or processors with discrete memory areas then communication bandwidth becomes a key issue. The provision of high speed communication protocol, rather than TCP/IP, may also provide increased inter-process communication speed. On a single machine handling multiple processes further optimisation of inter-process communication can be handled in a number of ways:

- Message passing amongst local processes maybe performed using shared memory. Providing data flow analysis indicates that the passing process makes no further use of the data structure being passed then shared memory provides a mechanism that allows processes on the same machine to share data without having to linearise or copy the data, This implementation of inter-process communication allows messages to be passed locally via shared address space without the programmer knowing about this underlying implementation difference. A good implementation should provide a system with no difference in operational semantics.
- The use of a copying garbage collector may automatically linearise data structures as they're being copied.

Message passing is an inherently unreliable mechanism, whether a process is local or external to the message sender. It is intended that this unreliability is handled by the programmer [7]. This type of recommendation is one which leads to numerous different solutions to the same problem and maybe an area that needs to be addressed in future releases of Erlang.

Future work for the compiler includes extending the process management system to take advantage of lightweight process (thread) support available in some operating systems. This would enable the runtime environment to overcome operating system limits such as the Unix process limit and provide a straight forward mechanism to implement more rapid communication between processes.

5 Compiler Performance

A number of programs are commonly used to benchmark Erlang implementations [8]. These grograms include:

1. Fibonacci computation - integer based benchmark, LCO.
2. Naive list length - list based benchmark, LCO.
3. Naive list reversal - list based benchmark.
4. Smith Waterman string matching algorithm.
5. Quicksort - list based benchmark, LCO.
6. Takeuchi - recursive computation, LCO.

The Erlang code for these benchmarks maybe found with [15]. The comparative compilation speeds run on a Pentium-100 with a Linux kernel 2.0.35 are shown in 4. Gerl is significantly faster for computationally intensive programs such as Takeuchi and Fibonacci, it performs poorly for programs that do significant amount of head-appending to lists such as naive list reversal and quicksort. Future work on handling list elements and associated garbage collection could significantly improve list performance.

	Gerl	JAM 4.6.4
fib.erl	9	19
tak.erl	29	59
nrev.erl	34	17
smith.erl	6	7
length.erl	14	68
qsort.erl	7	4

Figure 4: Selected Benchmarks (secs)

	Gerl	JAM 4.6.4
fib.erl	16	16
tak.erl	16	16
nrev.erl	19	17
smith.erl	39	21
length.erl	16	16
qsort.erl	22	17

Figure 5: Compilation Speeds (secs)

5.1 Compilation Speed

A compiler is intended to speed the performance of programs that are run many times; providing an overall time savings in delivering a service. Prototype developments are often discarded after limited testing; in these environments the speed of the compiler becomes an issue to consider.

Compiling via C and then compiling the C intermediate code and using **GNU gcc** as the second stage of a compiler has the potential to be very slow. Despite this Gerl performs reasonably well in compilation speed comparisons with the standard Erlang (bytecode) compiler. While the compiler is slower for larger pieces of code; small pieces of code take around the same time to compile. Figure 5 shows the compilation speed of Gerl/egcs-1.0.2 compared with the JAM 4.6.4 system on a Pentium-100 with a Linux 2.0.35 kernel, the times shown are the total time for 10 compilations of the listed program.

6 Compiled Object Size

A disadvantage of a compiling code is that native object code typically takes up 3-6 times more

	Gerl	Gerl C	JAM 4.6.4
fib.erl	10128	2609	1780
tak.erl	10744	2884	1606
nrev.erl	11712	4275	2014
smith.erl	24696	17294	3531
qsort.erl	14024	6485	2386

Figure 6: Compiled Object Sizes (bytes)

space than the equivalent bytecode compilation. Comparative sizes on an Intel x86 architecture, using **GNU gcc** version egcs 1.0.2 are given in Figure 6. Significant blows out in size occur with smaller files, primarily because **GNU gcc** includes a large amount of symbolic information. It should be noted that the minimum memory footprint of the Erlang JAM runtime system is significantly higher (4 times) than for the compiler's runtime system.

Work has been done on a combination approach to compilation [9]. With only selected functions being compiled into native code. This requires that native code work with the emulated code stack particularly for the organisation of values and errors.

7 Conclusion

The compiler provides a high performance Erlang engine which is tightly coupled with the Unix environment. It implements a significant portion of the Erlang standard. The compiler generates intermediate C code from Erlang source. It supports the following features: dynamic loading, run-time garbage collection, multiple process support over multiple machines, and type checking where possible. A key feature of the compiler is its automatic instrumentation of application code to enable continuous process monitoring.

The compiler is an open source development project and is available for public examination [15]. The benchmarking programs are available from the same location.

References

[1] Jonas Barklund. *ERLANG - the specification.* Ericsson, 1997.

[2] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, pages 197–206, 1993.

[3] William Clinger. *Proper Tail Recursion and Space Efficiency. Proceedings of ACM PLDI 1998*, 1998.

[4] R. Stallman et al. *GNU C Compiler - 2.7.1.* Technical report, Free Software Foundation, 1995.

[5] Pieter Hartel. The pseudoknot functional benchmark. *The Research Journal: Declarative Systems and Software Engineering Group*, 1995/96.

[6] Fergus Henderson, Thomas Conway, and Zoltan Somogyi. *Compiling Logic Programs to C using GNU C as a Portable Assembler.* Technical report, University of Melbourne, 1998.

[7] M. Williams J. Armstrong R. Virding. *Concurrent Programming in ERLANG.* Prentice-Hall, 1993.

[8] Erik Johansson and Christer Jonsson. *Native Code Compilation for Erlang.* Technical report, Computer Science Department, Uppsala University, 1996.

[9] Erik Johansson, Christer Jonsson, and Thomas Lindgren. *A pragmatic approach to compilation to Erlang.* Technical Report TR No. 136, Computer Science Laboratories, Ericsson Telecom AB, 1997.

[10] Simon Peyton Jones, Thomas Nordin, and Dino Oliva. C−−: A portable assembly language. In *1997 Workshop on Implementing Functional Languages.* Springer Verlag, 1997.

[11] Simon Marlow and Philip Wadler. *Ertc, A Type Checker for Erlang.* Technical report, Ellemtel Telecommunications Laboratories, Sweden, 1996.

[12] Richard M. Stallman and Roland H. Pesch. *Debugging with GDB: The GNU Source Level Debugger.* Technical report, Free Software Foundation, 1998.

[13] Paul Wilson. Uniprocessor garbage collection techniques. In *1992 International Workshop on Memory Management.* Springer Verlag, 1992.

[14] Geoff Wong. *A Software Monitoring Framework.* In *Erlang User Conference 1998*, Stockholm, 1998.

[15] Geoff Wong. *Gerl.* ftp://moo.cs.rmit.edu.au/pub/gerl.tgz, 1998.