A High Performance Erlang System

Erik Johansson
Computing Science Department, Uppsala University, Sweden
happi@csd.uu.se

Mikael Pettersson
Computing Science Department, Uppsala University, Sweden
mikpe@csd.uu.se

Konstantinos Sagonas
Computing Science Department, Uppsala University, Sweden
kostis@csd.uu.se

ABSTRACT

Erlang is a concurrent functional programming language designed to ease the development of large-scale distributed soft real-time control applications. It has so far been quite successful in this application domain, despite the fact that its currently available implementations are emulators of virtual machines. In this paper, we improve on the performance aspects of Erlang implementations by presenting HiPE, an open-source native code compiler for Erlang. HiPE is a complete implementation of Erlang, offers flexible integration between emulated and native code, and efficiently supports features crucial for Erlang's application domain such as concurrency. As our performance evaluations show, HiPE is currently the fastest among all Erlang implementations.

1. INTRODUCTION

The concurrent functional programming language Erlang was designed by Ericsson to address the needs of large-scale distributed soft real-time control applications [2]. Such applications routinely arise in products developed by the telecommunications industry. Erlang caters for these needs with a run-time system that provides many features often associated with an operating system rather than a programming language. These features include scheduling of lightweight concurrent processes, automatic memory management, networking, protection from deadlocks and programmer errors, and support for continuous operation even when performing software upgrades.

After around a decade of existence, Erlang is generally considered as a "success-story" in declarative programming languages; see e.g. [25]. Users experience that Erlang allows telecommunication systems to be programmed with less effort and fewer errors than by using conventional programming language technology [1, 5]. It is worthwhile to note that such systems typically consist of several hundred thousand lines of source code (the size is partly due to the complexity of the telecommunication protocols), and rely heavily upon the concurrency capabilities of Erlang.

The industry, besides Ericsson, is showing a growing interest in Erlang, but there is a very limited choice of compilers, partly due to Erlang's—until recently exclusive—"in-house" development. Also, as an implementor of these compilers publicly admits [1]: 'performance has always been a major problem' and 'we are (even) considering adding imperative features to the language to solve these (performance) problems'. Indeed, the performance of current implementations of Erlang is inferior to that of good implementations of other functional programming languages; see also [8, 11]. In the competitive market of telecommunications, however, the need for a high-performance implementation is sometimes pressing.

As one such example, consider AXD 301, a new generation ATM switching system from Ericsson [5]. The major part of AXD 301's software is written in Erlang; it consists of about 480,000 lines of Erlang code, with about 95,000 of them constituting the time-critical modules of the system. Speeding up this time-critical part would be more than welcome by the AXD 301 engineering team, let alone Ericsson, because this speedup directly corresponds to the ATM switch being capable of servicing more requests; see also [5].

Currently, complete implementations of Erlang are based on emulators of virtual machines. This gives them good portability, but emulation incurs a performance penalty to Erlang programs which some users wish—and in some cases need to—avoid. Ways to avoid the performance problems caused by emulation are: 1) compile to a sufficiently low-level and fast language such as C or 2) use the recently proposed `C--` [19] as a portable assembly language, 3) use a retargetable code generator such as ML-RISC [18] or 4) the `gcc` back-end [21], or 5) compile directly to native code. Each of these implementation choices has well-known pros and cons but one can roughly expect a decrease in portability and an increase in performance and implementation effort for a higher choice number; see also the above references and the references therein.

Perhaps another issue deserves attention: byte-code emulators usually result in smaller object code size than C-based or native code compilers. Although object code size is becoming less and less of a concern nowadays, it is still a potential problem when the source code of the application consists of several hundred thousand lines. Also, not paying attention to code size can result in poor I-cache behaviour leading to significant performance degradation. Ideally, a system should provide a seamless integration of emulated and native code execution, and allow its user to choose the execution mode individually for each application component based on the various space/time trade-offs that are involved. Information about these trade-offs should also be something that the system provides to its user.

This paper presents our approach to the efficient execution of Erlang. We have developed a system, HiPE, which combines the performance characteristics of a native code compiler with the benefits of an emulated implementation. HiPE currently uses the JAM emulator [1] as a basis and allows selective compilation of whole modules or individual functions into native code, which is then executed directly by the underlying hardware. Besides fully describing the architecture of HiPE and the changes to the JAM run-time system needed to support native code execution, we discuss various technical issues that this emulated/native code integration entails in the context of Erlang and how we dealt with them. More specifically:

- we pay special attention to supporting *hot-code loading* (see next section) and error handling in HiPE;
- we describe a method for performing tail call optimization in a mixed mode of execution where a separate stack is used for each mode—this method is probably folklore but, to the best of our knowledge, it has not been reported in the literature;
- we compare and analyse the performance of existing Erlang implementations on "standard" small benchmarks and on large programs from actual industrial Erlang applications and show that HiPE's performance is superior.

To make this paper self-contained, we begin by reviewing the characteristics of Erlang (Section 2). Sections 3 and 4 form the main part of this paper and describe the basic characteristics of HiPE, its architecture, and the integration of native and emulated execution within the same run-time system. Section 5 contains a performance comparison of HiPE against other implementations of Erlang, both industrial and academic, and against other functional languages. Section 6 contains additional measurements on some of the benchmark programs used in our performance evaluation that give more insight on HiPE's performance. We end this paper with some concluding remarks.

2. A BRIEF INTRODUCTION TO ERLANG

Erlang is a dynamically typed, strict, concurrent functional programming language. It is possible to create closures in Erlang, but typical Erlang programs are mostly first-order. Erlang's basic data types are atoms, numbers (integers with arbitrary precision and floats), process identifiers, and references; compound data types are lists and tuples. There is no destructive assignment of variables and the first occurrence of a variable is its binding instance. Function selection happens using pattern matching. Erlang's design inherits some ideas from concurrent constraint logic programming languages such as the use of flat guards in function clauses.

Processes in Erlang are extremely light-weight, their number in typical applications is quite big, and their memory requirements vary dynamically. Erlang's concurrency primitives—`spawn`, "`!`" (send), and `receive`—allow a process to spawn new processes and communicate with other processes through asynchronous message passing. Any data value can be sent as a message and processes may be located on any machine. Each process has a *mailbox*, essentially a message queue, where each message sent to the process will arrive. Message selection from the mailbox occurs though pattern matching. There is no shared memory between processes and distribution is almost invisible in Erlang. To support robust systems, a process can register to receive a message if another one terminates.

Erlang applications typically consist of a number of modules: an Erlang module defines a number of functions. Only explicitly exported functions may be called from other modules. Calling functions in different modules, called *remote calls*, is done through supplying name of the module of the called function. During execution of functions, tail call optimization is performed. As in other functional languages, Erlang's memory management is automatic through garbage collection. The real-time concerns of the language call for bounded-time garbage collection techniques; see [24, 16]. In practice, garbage collection times are usually small as most processes are short-lived or small in size.

To perform system upgrading while allowing continuous operation, an Erlang system needs to cater for the ability to change the code of a module while the system is running, so called *hot-code loading*. Processes that execute old code can continue to run, but are expected to eventually switch to the new version of the module by issuing a remote call (which will always invoke the most recent version of that module). Erlang provides mechanisms for allowing a process to timeout while waiting for messages and a catch/throw-style exception mechanism for error handling.

The Erlang language was purposely designed to be small, but comes with libraries containing a large set of *built-in functions* (known as *BIFs*). With the Open Telecom Platform (OTP) middleware [23], Erlang is extended with standard solutions to common requirements of telecommunication applications (servers, state machines, process monitors, load balancing), standard interfaces (CORBA), and standard communication protocols such as HTTP and FTP.

3. JAM: THE BASIS OF HIPE

HiPE is based on the bytecode emulated JAM implementation of Erlang, to which it adds the ability to compile and execute Erlang as native machine code. HiPE is a new component (currently 30,000 lines of Erlang code and 3,000 lines of C and assembly code) added to an otherwise mostly unchanged JAM system; only the JAM emulator and the garbage collector have been extended to be aware of native code. Because of this tight integration, we describe relevant aspects of the basic JAM system here; Section 4 continues with HiPE-specifics.[1]

[1]HiPE is publicly available as open-source. The current release is based on Ericsson's Open Source Erlang 47.4.1. See http://www.csd.uu.se/projects/hipe.

3.1 The JAM system

The JAM-based Erlang implementation uses JAM: a virtual stack machine whose primitive operations closely correspond to the Erlang language. For example, since Erlang is a dynamically-typed language, JAM uses a self-describing data representation with tagged values [10], and the emulator's instructions operate on tagged values, never on raw machine values.

3.1.1 The JAM compiler

The JAM compiler is a non-optimising compiler which performs a straightforward translation to the JAM virtual stack machine. The object files contain bytecodes and relocation entries which describe all symbolic references that must be resolved by the loader.

3.1.2 The JAM loader

The JAM loader translates JAM bytecodes from the external format to the internal format expected by the JAM emulator.

Erlang atoms are symbolic constants, like atoms in Prolog or symbols in Lisp. The internal representation of an atom is its position in the atom table, which is not known until runtime. Therefore, the first task of the loader is to replace symbolic references to atoms by their current internal representation.

A remote (non-local) function call is represented by a triple (module name, function name, function arity). First the names are translated to internal atoms. Then special cases are identified, such as calls to the `erlang` module which become calls to C functions in the runtime system. Finally, the JAM instruction at this call site is patched to reflect the result.

After a module has been loaded, a global symbol table is updated with information about the module, its exported functions, and the code addresses at which those functions start.

3.1.3 The JAM emulator

The JAM emulator is a single C function which executes JAM instructions represented as bytecodes. The JAM implementation is indirectly threaded [4, 17]. The emulator is invoked on a runnable Erlang process, and executes code for that process until it blocks. A process blocks either when it attempts to read a message and its message queue is empty, or when its "time slice" has expired. Time slices are represented by work budgets, which are explicitly decremented and checked at specific points in the emulator.

Each Erlang process is described by a process control block (PCB), a stack, a heap, and a set of pointer registers:

- `sp` next word on the stack
- `fp` start of current function's activation record
- `ap` first actual parameter
- `pc` current bytecode instruction
- `cc` debug information about the current function

The stack discipline is simple but unoptimised. At a call, the parameters are pushed in left-to-right order, followed by a 4-word continuation record containing the caller's `fp`, `ap`, `pc`, and `cc`. Then `fp` is set to point to the start of this record, `sp` to the first word after the record, and `ap` to the first parameter (derived from `fp` and the callee's arity); see Figure 1. (Note that `ap` now has the same value as the caller's `sp` had before the call.) At return, `sp` is reset to `ap`, then `cc`, `pc`, `ap`, and `fp` are restored from the frame, and the return value is pushed onto the stack.

	...	
`ap` →	parameter #1	
	...	
	parameter #n	
`fp` →	caller's `fp`	
	caller's `ap`	
	caller's `pc`	
	caller's `cc`	
`sp` →	...	(stack grows down)

Figure 1: JAM stack on entry to a function

In JAM, tail calls are complicated by the fact that the four-word continuation frame is pushed *after* the parameters instead of before. Since the continuation frame is adjacent to the parameter area, it must be relocated whenever there is a tail call and the caller and callee have different number of parameters. At a tail call, the JAM emulator copies the frame into temporary variables, then copies the outgoing parameters from the bottom of the stack to the parameter area, and then (if necessary) moves the copy back to the stack.[2]

Exception handling is implemented by *dynamic tracking* [3]. On entry to a protected code block, a 2-word catch frame is pushed onto the stack, containing a pointer to the previous catch frame and the address of the first instruction after the protected block. The address of this frame is saved in the PCB. To raise an exception, the stack is unwound one call-frame at a time, until the activation record containing the current catch frame is found. The unwinding process simultaneously restores the `sp`, `fp`, `ap`, and `cc` registers.

3.2 Processes and memory management

An Erlang *node* is an instance of the Erlang runtime system executing on a given machine. On Unix, this is a single Unix process. Within a node, Erlang processes are created dynamically and execute as coroutines. A C procedure acts as scheduler, continuously selecting a runnable process and passing it to the emulator for execution.

Each Erlang process has a PCB, a stack, and a *private* heap for the data structures it creates. An Erlang process starts with small stack and heap areas, which are grown when needed. Compared to a typical implementation of Posix threads in Unix, which would allocate in the order of one megabyte of virtual memory for each thread's stack, Erlang

[2] Performance could be improved by shrinking the frame to a minimum: `ap` is redundant and `cc` can be computed from `pc` when needed. If the continuation was pushed *before* the parameters, it could remain in place during tail calls [7, Section 4.6.1].

processes are extremely lightweight. An Erlang node is expected to handle hundreds or thousands of Erlang processes with relative ease.

The garbage collector is of a standard two-generational stop-and-copy type [16]. It does, however, have one interesting design feature: each process' heap is strictly private to that process, and no references are allowed from one process to another's heap. The advantage of this arrangement is that it simplifies memory management. Since references from other processes cannot occur, garbage collection is a strictly local activity, and when a process terminates, its memory can be reclaimed immediately. This is believed to reduce total memory management costs, since the majority of Erlang processes are expected to be short-lived or to have small amounts of live data. The disadvantage, however, is that message passing must be implemented by data copying, which reduces sharing and increases the pressure on the memory caches.[3] Messages are expected to be small, however.

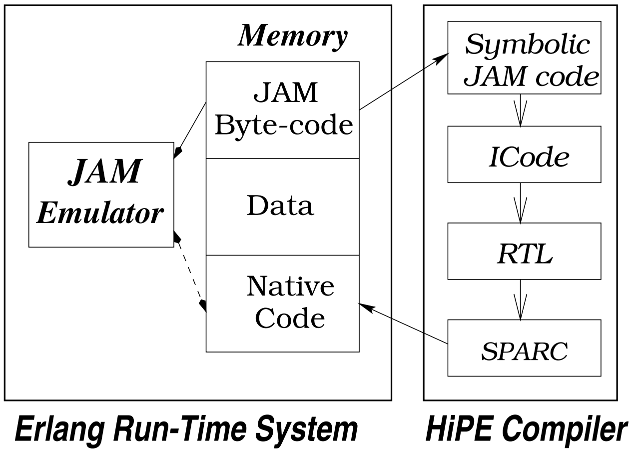

Figure 2: Intermediate representations in HiPE.

4. HIPE: SYSTEM OVERVIEW

The HiPE compiler is called as an ordinary Erlang function, within a running Erlang system. The smallest unit of compilation is a function: given the name of an existing Erlang function, the compiler translates that function's JAM bytecodes to SPARC V9 machine code, and then a linking phase updates the system state so that future calls invoke the native code; see Section 4.2. In this respect, HiPE resembles a (user-invoked) JIT compiler. Alternatively, the HiPE compiler can compile a whole Erlang module, and the result of the compilation can be saved in a file, as symbolic SPARC code. This file can later be loaded explicitly into the HiPE system.

Some Erlang applications are delivered only in bytecode format and no sources are available. For example, before Erlang became open-source, Erlang libraries were only present as JAM bytecode files. Therefore, HiPE was designed to be able to take already-loaded functions in JAM bytecode format instead of Erlang source code as input. Although this means that HiPE cannot perform some high-level optimizations, this compilation scheme offers more freedom to its users: users have the possibility to identify (using HiPE's profiling tools) those functions and call paths that would benefit most from compilation to native code and selectively compile them. This way, HiPE combines the performance of a native code system with the code compactness of a bytecode system. This integration is very tight: code compiled by HiPE uses the same runtime system and the same built-in functions as the JAM emulator.

4.1 The HiPE compiler

The compiler has four intermediate representations: an internal representation of JAM bytecode, a high level intermediate language (ICode), a general register transfer language (RTL), and a machine-specific assembly language, currently only SPARC; see Figure 2. ICode, RTL, and SPARC are represented as control flow graphs of basic blocks.

[3]We are planning to measure memory subsystem performance by implementing a shared-heap runtime system and comparing it against the current runtime system.

The JAM bytecodes are translated to symbolic form by a straightforward process. Internal atom numbers are converted to real atoms, and branches to local functions are translated to call instructions with symbolic function names.

ICode is based on a register-oriented virtual machine for Erlang. Arguments and temporaries are located in an infinite number of registers, and all values are proper Erlang terms. The call stack is implicit, and calls preserve registers. Bookkeeping operations, such as heap overflow checks, context switching, and time-slice decrements, are implicit. The translation from JAM bytecode to ICode uses a simulated stack to map JAM stack slots to ICode registers. To simplify dependency analysis in later compilation passes, a renaming post-pass ensures that independent live ranges use different registers.

The ICode is then optimised with standard compiler optimizations such as copy and constant propagation, and constant folding. This is done in one pass over all extended basic blocks. Dead code removal is then performed to remove assignments to dead temporaries.

Unreachable code is removed by the translation to RTL, since only reachable basic blocks are inserted in the RTL control flow graph. Operations on Erlang values are expanded to make data tagging and untagging explicit.

The optimisations that were performed on ICode are applied again to the RTL code. Heap overflow tests, call stack management, and the saving and restoring of registers around calls are made explicit, and the standard optimisations are applied again. In order to limit the number of heap overflow tests, they are propagated backwards as far as possible, and adjacent tests are merged.

The RTL code then is translated to abstract SPARC code, and registers are assigned using a graph-colouring register allocator similar to the one of Briggs *et al.* [6]. Finally, symbolic references to atoms and functions are replaced by their values in the running system, memory is allocated for the code, and the code is linked into the system.

4.2 The HiPE linker

As described before, Erlang requires the ability to upgrade code at runtime, without affecting processes currently executing the old version of that code.

The JAM system maintains a global table of all loaded modules. Each module descriptor contains a name, a list of exported functions, and the locations of its *current* and *previous* code segments. The exported functions always refer to the current code segment. At a remote function call, `module:function(parameters...)`, the JAM emulator first performs a lookup based on module and function name. If the function is found, the emulator starts executing its bytecodes. Otherwise, an error handler is invoked.

In native code, each function call is a direct machine-level call to an absolute address. When the caller's code is being linked, the linker initialises the call to directly invoke the callee. If the callee has not yet been loaded, the linker will instead direct the call to a stub which performs the appropriate error handling. If the callee exists, but only as emulated bytecode, the linker directs the call to a stub which in turn will invoke the JAM emulator.

In order to handle hot-code loading and dynamic compilation at runtime, the linker also maintains information about all call sites in native code. This information is used for *dynamic code patching*, as follows:

- When a module is updated with a new version of the emulated code, all remote function calls from native code to that module are located. These call sites are then patched to call the new emulated code, via new native-to-emulated code stubs.
- When an emulated function is compiled to native code, each native code call site which refers to this function is patched to call the new native code. The first instruction in the bytecode is also replaced by a new instruction which will cause the native code version to be invoked. Finally, the native-to-emulated stub used to invoke it from native code is deallocated.
- When a module is unloaded and its memory is freed, all native code call sites referring to this module are patched to instead invoke an error handling stub. All native code call sites within this now non-existent module are also removed from the linker's data structures, to prevent future attempts to update them.

Both the standard Erlang system and HiPE support load-on-demand of modules. When invoked, the error handler for undefined function calls will attempt to load the JAM bytecodes for that module from the file system. If this is successful, the call continues as normal. As a side-effect of loading the JAM module, the HiPE linker will patch native code call sites as described above.

4.3 Native code calling conventions

In the HiPE runtime system, an Erlang process can execute both emulated JAM code and native SPARC code. To facilitate data sharing, HiPE uses the same data representation as JAM. However, the JAM calling convention is inappropriate for native code, since JAM passes all parameters on the stack and uses large call frames containing redundant information. Instead, native code passes the return address and the first five parameters in registers, remaining parameters (if any) on the native stack, and shrinks the fixed portion of stack frames to a single word for preserving the previous return address. Currently, HiPE does not use the SPARC's register windows; instead, registers are saved and restored as needed around function calls.

HiPE uses two stacks for each process, one for emulated code (the *estack*) and one for native code (the *nstack*). An earlier version of HiPE used only one stack, but that scheme was quickly abandoned as it was found to be quite complex and difficult to implement correctly. Our current dual-stack approach has disadvantages and advantages too:

– As described in Section 3.1.3, the JAM emulator implements exception handling by creating a linked list of catch frames on the stack. Native code uses the same strategy, which means that each stack may contain pointers to the other. If the runtime system relocates one stack (to increase its size), then the other stack must be traversed so that the catch frame links can be updated.

\+ By separating the stacks the stack-scanning code in the garbage collector is kept simple. With a single-stack approach, the scanning code would have to know when to switch "mode", in order to correctly deal with the different stack frame layouts. This is certainly doable, but would require more effort to implement correctly.

Foreign Function Interface

Erlang programs can and often do call C functions: standard procedures (BIFs) in the runtime system are written in C, as are the I/O modules. For each BIF, there is a machine-code stub which is directly callable from native code. The stub takes care of saving the native-code state registers before invoking the C function on the C stack. The stub also checks for exceptions at return, and either returns to its caller or invokes the current Erlang exception handler.

However, the Erlang runtime system is not designed to permit arbitrary recursive calls between Erlang and C functions. Whether the Erlang code is emulated or native makes no difference. A detailed explanation is beyond the scope of this paper, but the limitation stems mainly from the non-reentrancy of the Erlang process scheduler (a C function) and the way in which the runtime system periodically polls I/O channels. (It would not be overly difficult to modify the runtime system to eliminate this limitation, however.)

4.4 Mode-switching

In HiPE, a *mode-switch* occurs whenever there is a transfer of control from native code to emulated code, or vice-versa. We made the design decision that the mere presence of multiple execution modes should not impose any runtime overheads, as long as no mode-switches occur. This design requirement calls for great care when implementing mode-switches, not only for performance, but also for correctness.

4.4.1 *Where do switches occur?*

Since HiPE compiles individual functions to native code, a mode-switch must occur whenever there is a flow of control from one function to another, and the two functions are in different modes. Thus, mode-switches occur at call and return sites. Erlang's exception mechanism also introduces mode-switches, viz. when an exception is thrown from code executing in one mode, and the most recent catch handler is in a different mode. We will refer to these cases as *call*, *return*, and *throw* events, respectively.

4.4.2 *When do switches occur?*

How does the system discover that a particular instance of a call, return, or throw event must perform a mode-switch?

Call Events

Older Lisp systems often use a dynamic test at calls to determine the mode of the callee (e.g. compiled or emulated), and then perform the appropriate action.

Another common choice is to use a fixed mode for calls, usually native code. Emulated functions are represented as small native code stubs which invoke the emulator when called. The advantage of this approach is that no dynamic type test is needed when both caller and callee are in native code. The disadvantage is that calls between emulated-mode functions are penalised since they have to make conversions to and from the native-code calling conventions.

HiPE uses a pseudo-static approach in which calls always use the mode of the caller. As described in Section 4.2, if a native-code caller refers to an emulated-mode callee, then the linker redirects the call instruction to instead invoke a native-code stub, which in turn causes a switch to emulated mode. If an emulated function is compiled to native code, then the start of the original bytecodes is overwritten with a special emulator instruction which causes a switch to native mode. (The asymmetry between these cases is due to the fact that the HiPE linker only has knowledge about call sites in native code.)

Return Events

Whenever a recursive function call causes a mode-switch, the return sequence must be augmented to perform the inverse mode-switch. Like calls, this is often implemented by dynamic tests, a fixed-mode convention, or by same-mode stubs.

If dynamic tests are used, then the return address is tagged to indicate that a mode-switch is required when the callee returns. Typically, either the lowest or the highest bit is used. The lowest bit is available if return addresses are aligned on some $2^i > 1$ boundary. Using the highest bit imposes a limit to the usable address space.

If a fixed-mode convention is used for calls, then the same mode is also used for returns. If that mode is native code, then a call from an emulated-mode function must push a "stub" native-mode return address which causes control to flow back into the emulator.

HiPE uses a same-mode convention for returns. When a call causes a mode switch, a new continuation (stack frame) is created in the mode of the callee. The return address in this continuation points to code which causes a switch back to the caller's mode. For returns from native to emulated code, the return address points to machine code in the runtime system. For returns from emulated to native code, the return address points to a special emulator instruction. We made this choice in HiPE because it causes no overhead except during mode-switches, and it minimised the amount of changes we had to made to the existing JAM emulator.

Throw Events

HiPE deals with exception throws in the same way as it deals with function returns: a same-mode convention augmented with mode-switching stack frames. When a call causes a mode-switch, a new exception catch frame is created in the mode of the callee. The handler address in this catch frame points to code which switches back to the caller's mode, and then re-throws the exception. Thus, when a call causes a mode-switch, *two* frames are pushed: first a catch frame, then a return frame. The code at the return address in the return frame knows that it also has to remove the catch frame beneath it before switching mode.

4.4.3 *Maintaining tail-recursion*

The same-mode calling convention with mode-switch stack frames is efficient and easy to implement. For many programming languages, this would be enough. However, like most other functional programming languages, Erlang relies on tail-recursive function calls for expressing iteration. Consider the following sequence of tail calls, where each f_i^e is an emulated function, and each f_j^n is a native code function:

$$f_1^e \stackrel{tail}{\to} f_2^n \stackrel{tail}{\to} f_3^e \stackrel{tail}{\to} f_4^n \stackrel{tail}{\to} \cdots$$

A correct implementation of Erlang is expected to execute such a sequence in bounded stack space, regardless of its length. Unfortunately, at each call, a new mode-switch stack frame is pushed, to make the return perform the inverse mode-switch. Thus, stack space usage will grow linearly with the length of the sequence of tail calls, and tail-recursion optimisation is lost.

HiPE solves this problem as follows. The return address in a mode-switch stack frame will always have a known value: either the address of the return mode-switch routine (in native mode), or the address of the return mode-switch instruction (in emulated mode). Thus, a simple runtime test is able to distinguish mode-switch stack frames from normal stack frames. Now, consider the following call sequence:

$$f^e \to g^n \stackrel{tail}{\to} h^e$$

When f^e calls g^n, it pushes two mode-switch frames on the native-code stack: first a catch frame, then a return frame. When g^n tail calls h^e, the system would normally push two new mode-switch frames, on the emulated-code stack. Instead, HiPE implements a mode-switch call event as follows:

1. if the current return frame is a mode-switch frame, then
   (a) remove the mode-switch return frame from the caller's stack

(b) remove the mode-switch catch frame from the caller's stack

(c) invoke the callee;

otherwise

2. push a mode-switch catch frame on the callee's stack
3. push a mode-switch return frame on the callee's stack
4. invoke the callee

By preventing adjacent mode-switch frames from being created, the test restores proper tail-recursive behaviour. The test itself is not expensive, and is only executed when there is a mode-switch call. This implementation refutes a statement in [9, Section 4.4.3], where it was claimed that the use of mode-switch stack frames loses tail-recursion optimisation. Although similar methods to maintaining proper tail-recursion in the context of mixed mode execution have been used in some Prolog implementations (e.g. BIM Prolog, SICStus Prolog) and perhaps elsewhere, these methods are probably folklore: to the best of our knowledge, they have never been in print.

4.4.4 Mode-switching in suspend/resume events

In addition to the call, return, and throw events described above, HiPE may also need to perform mode-switches when a process is suspended or resumed.

The scheduler in the Erlang runtime system has no knowledge about the current mode of a process. It assumes, implicitly, that each process is executed by the JAM emulator. Therefore, when a process is created or resumed, the scheduler simply passes the process' PCB to the JAM emulator for execution.

When a process is suspended while executing in native code, HiPE sets the resume address in the PCB to point to a special emulator instruction. When the scheduler resumes the process, the JAM emulator executes this instruction, which transfers control to the suspended native code.

4.5 Modifications to the JAM emulator

As described in Sections 4.2–4.4, we have modified the JAM emulator to support mixing native and emulated code. In summary, these modifications are:

- The JAM loader registers the location of each function's bytecodes with the HiPE linker.
- A native-code stack has been added to the PCB, together with a few native-code variables (stack pointer, resume address).
- The garbage collector has been extended to scan the native-code stack, and to repair catch frame links when either stack is relocated.
- A small number of instructions have been added to the JAM emulator, to support mode-switching between emulated and native code.

A previous version of HiPE used dynamic tests in the JAM emulator instead. At each call, a check was made if the target also had a native-code version, and at each return, throw, and resume, a check was made if the return address was zero, which was interpreted as a signal to switch mode. That design required changes to many different locations in the emulator, complicated the mode-switch stack frame management, imposed runtime overheads on emulated code, and was generally ugly and difficult to maintain.

In contrast, our current design requires only a small localised extension of the emulator, and imposes no runtime overheads except during mode-switches.

4.6 Performance instrumentation features

To support performance analysis and benchmarking, and to help users identify which parts of their code could benefit most from compilation to native code, we have added two kinds of performance instrumentation features to HiPE's runtime system: *software event counters* and *hardware performance counters*.

The software event counters keep track of how often various interesting operations are performed. These include: the number of times each Erlang function is called, the number of times each built-in library function is called, how many times each JAM instruction is executed, and how many times control is passed between emulated and native code.

The hardware performance counters are event counters inside the UltraSPARC processor[4]. They are used to measure the number of clock cycles spent in code regions, and to provide hardware-specific information, for example the amount of time lost due to stalls and cache misses. The reason for a stall can also be determined: data cache miss, instruction cache miss, external cache miss, or a branch misprediction. For more details on the instrumentation facilities, the reader is referred to [14].

5. PERFORMANCE EVALUATION

We conducted our performance comparison on a 143 MHz single-processor Sun UltraSPARC 1/140 with 128 MB of primary memory running Solaris 2.6. Although slow by today's standards, this machine was used because we could ensure it was lightly loaded during the benchmark runs, and it had enabled user-level access to the UltraSPARC's performance instrumentation facilities [22]. The latter was needed for the performance results presented in the following section. When using HiPE, all functions actually called in the programs were compiled to native code.

5.1 Erlang systems used in the comparison

Besides HiPE (version 0.90), three other Erlang systems were used in this comparison: JAM, BEAM, and Etos. The JAM and BEAM systems used in our measurements are from Ericsson's Erlang 47.4.1 (upon which HiPE 0.90 is based). The version of Etos used is 2.3.[5] We describe BEAM

[4] Many modern processors have hardware performance counters, although their functionality and the level of OS support varies.

[5] Versions of Etos & HiPE used in [8] are significantly older than those used here.

and Etos below; HiPE and JAM have been fully described earlier in this paper.

BEAM

The BEAM [12] is a register-based abstract machine, influenced by the Warren Abstract Machine (WAM) [26] used in many Prolog implementations. Compared to JAM, the translation of Erlang code to BEAM abstract machine instructions is more advanced. For example, the treatment of pattern matching is better in the BEAM system, even though a full pattern match compiler (like that in e.g. [20]) is not implemented. Also, BEAM uses a direct-threaded emulator [4, 17] using `gcc`'s first-class labels extension [21]: instructions in the abstract machine code are addresses to the part of the emulator that implement the instruction.

Etos

Etos, described in [8], is a system based on the Gambit-C Scheme compiler. It translates Erlang functions to Scheme functions which are then compiled to C. The translation from Erlang to Scheme is fairly direct. Thus, taking advantages of the similarities of the two languages, many optimizations in Gambit-C are effective when compiling Erlang code. Among these optimizations are inlining of function calls (currently only *within* a single module) and unboxing of floating-point temporaries. Etos also performs some optimizations in its Erlang to Scheme translation, for example, simplification of pattern-matching. Process suspension in Etos is done using `call/cc` implemented using a lazy copying strategy; see [13]. When a process is suspended, the stack is "frozen" so that no frame currently on the stack can be deallocated. When control returns to a suspended process, its stack frames are copied to the top of the stack. When the stack overflows, the garbage collector moves all reachable frames from the stack to the heap. In general, suspending and resuming a process will require its stack to be copied at least once. In contrast, the JAM/BEAM/HiPE runtime systems handle processes explicitly; saving or restoring the state of a process involves storing or loading only a small number of registers. The Etos compiler is work under progress, and it is not yet a full Erlang implementation. We have therefore been able to run only relatively small benchmarks on Etos.

5.2 Time on benchmark programs

We start our performance comparison using the following set of "standard" small sequential benchmarks also used in [8]:

fib A recursive Fibonacci function. Calculates `fib(30)` 50 times.

huff Huffman encoder. Compresses and uncompresses a short string 5000 times.

length A tail-recursive list length function finding the length of a 2000 element list 100,000 times.

nrev Naive reverse of a 100 element list 20,000 times.

qsort Ordinary quicksort. Sorts a short list 50,000 times.

smith The Smith-Waterman DNA sequence matching algorithm. Matches one sequence against 100 others; all of length 32. This is done 30 times.

Benchmark	HiPE	Etos	JAM	BEAM
fib	33.8	31.8	281.4	120.6
huff	11.9	12.1	234.7	69.2
length	22.7	17.2	375.6	98.9
nrev	18.5	24.4	241.3	56.9
qsort	12.3	11.0	208.1	97.6
smith	11.4	11.6	114.6	53.9
tak	13.5	12.8	140.1	100.2
decode	22.8	52.4	67.8	49.0

Table 1: Times (in secs) for small sequential benchmarks.

tak Takeuchi function, uses recursion and integer arithmetic intensely. Calculates `tak(18,12,6)` 1000 times.

and a medium-sized one ($\approx$ 400 lines):

decode Part of a telecommunications protocol. Decodes an incoming `binary` message 500,000 times.

Table 1 contains the results of the comparison. In all benchmarks, HiPE and Etos are the fastest systems: in small programs they are between 7 to 20 times faster than JAM and 3 to 8 times faster than the BEAM implementation; see also Figure 3. Excluding **length** and **nrev** where HiPE and Etos show complementary behaviour, the performance difference between these two systems on small programs is not significant. In **decode**, where it is probably more difficult for Etos to optimize operations and pattern matching on `binary` objects (i.e. on immutable sequences of binary data), HiPE is more than 2 times faster than Etos. HiPE is faster than JAM and BEAM, but not to the same extent as for the other benchmarks.

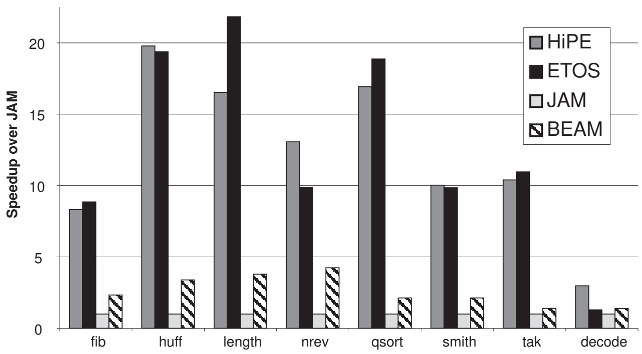

Figure 3: Speedup compared to JAM for small benchmarks.

Next, we compare the Erlang implementations on concurrent programs. As mentioned in the introduction, most Erlang programs rely heavily on the concurrency primitives of the language. Thus, these programs call for special attention in good Erlang implementations. The benchmark programs we used are:

ring Creates a ring of 10 processes and sends 100,000 messages. The benchmark is executed 100 times.

Benchmark	HiPE	Etos	JAM	BEAM
ring	37.1	76.0	101.6	72.5
stable	12.8	27.9	37.8	19.5
life	5.6	20.1	13.4	8.7

Table 2: Times (in secs) for small concurrent benchmarks.

stable Solves the stable marriage problem for 10 men and 10 women 5000 times.

life Executes 1000 generations in Conway's game of life on a 10 by 10 board where each square is implemented as a process.

Table 2 contains the results of the comparison. Once again, HiPE is the fastest system: it is around 2.5 times faster than JAM, 55% faster than BEAM (95% on **ring**), just over 2 times faster than Etos on **ring** and **stable** and more than 3.5 times on the **life** benchmark; see also Figure 4. In fact, Etos 2.3 does not seem to be significantly faster than JAM and is slower than BEAM when processes enter the picture. We suspect that Etos' implementation of concurrency via `call/cc` is not very efficient.

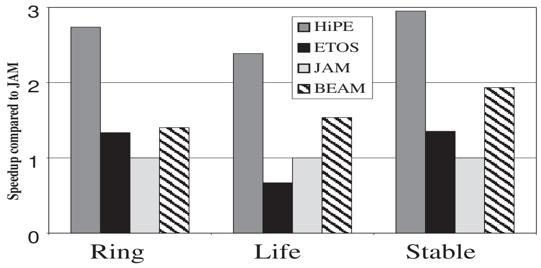

Figure 4: Performance speedup for concurrent benchmarks.

5.3 Time on real programs

To us, it was very unclear whether performance experiences gathered from the study of small or medium-sized benchmarks are applicable to real-life applications of Erlang. We thus also compared the performance of HiPE on quite large Erlang programs. The programs used in this endeavour were:

JAM Compiler This "application" is incestuous, but large nevertheless. The used portion of the compiler consists of 30 modules totalling $\approx 18,000$ lines of Erlang code. The benchmark is to compile 11 of these modules using the JAM compiler compiled in each of the systems.

Eddie An HTTP parser which handles 30 complex http-get requests. Excluding the OTP libraries used, it consists of 6 modules for a total of 1,882 lines of Erlang code. The benchmark is executed 1,000 times.

AXD/SCCT This is the time-critical software part of the AXD 301 switch mentioned in the introduction. Not counting standard libraries, it consists of about 95,000 lines of Erlang code. This actual benchmark of the ATM switch sets up and tears down a number of connections 100 times; 501 functions are used in the benchmark which amount to $\approx 15\%$ of the code. The remaining code provides other ATM services or handles errors that can occur in practice (but not during the benchmark run!). These parts were not compiled to native code.

Program	HiPE	JAM	BEAM
JAM Compiler	5.4	17.2	5.9
Eddie	18.8	93.6	40.0
AXD/SCCT	68.0	109.9	84.5

Table 3: Execution times (in seconds) for large programs.

Table 3 shows the results of this comparison.[6] HiPE is once again the fastest system. However, as we move from benchmarks to real-world applications of Erlang, programs tend to spend more and more of their execution time in built-ins from the standard library. For example, the **AXD/SCCT** program extensively uses the built-ins to access the shared database on top of the Erlang term storage; see the data presented in the Section 6 and in [14].

As the implementation of these built-ins is currently shared by all three systems, the percentage of execution spent in these builtins becomes a bottleneck and HiPE's speedup is less than before. Still, HiPE is 24% faster than BEAM on the largest benchmark, and considerably faster than the JAM implementation on which it is based; see also Figure 5.

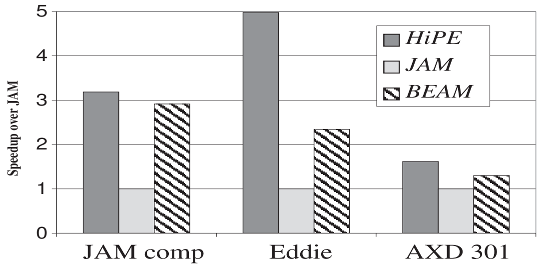

Figure 5: Performance speedup for large programs.

5.4 Code size

During benchmarking, all called functions of programs were compiled to native code. This makes perfect sense in small programs but might not be the best approach in all cases. As mentioned, users of HiPE can selectively compile time-critical parts of their applications to native code and use the emulator for the remaining parts. Still, HiPE's space overhead is not prohibitive and compilation of whole applications to native code is a valid option.

Table 4 provides some evidence to that effect. For JAM, BEAM, and HiPE it presents the size (in bytes) of the *loaded* code that is *used* by the benchmarks in this section. In all cases but **AXD/SCCT** all functions of loaded modules are

[6] Etos is not included here; it currently cannot handle these large programs.

Program	JAM	BEAM	HiPE	Etos†
fib	616	1937	4080	17808
huff	2530	7691	38728	90744
length	678	2085	4400	22424
nrev	750	2145	4932	24444
qsort	850	3049	6308	24784
smith	1416	4709	13616	71760
tak	656	2057	4712	23684
decode	1933	8641	14012	100132
ring	784	2737	6920	29380
life	1400	5069	15280	63772
stable	1299	4701	14008	59264
Eddie	21717	81075	170308	N/A
AXD/SCCT	83370	—	244936	N/A

Table 4: Code-size (in bytes) for the programs used.

used in the benchmark; in **AXD/SCCT** only a fraction of the functions is used. As the BEAM-based Erlang system does not provide any means of obtaining the size of individual functions (only of whole modules), we could not report the corresponding number for BEAM. For small programs, the code size increase of HiPE compared to JAM is between 7 to 15 times. Compared to BEAM, HiPE requires 2 to 3 times more code space (excluding **huff**). In large programs, the space overhead is on the low end; see e.g **AXD/SCCT**. For comparison, we also present the sizes of stripped `.o1` files produced by the Etos compiler; they are quite big partly due to the aggressive inlining performed by Etos 2.3. It is not clear to us to what extent Etos benefits from its use of inlining. Note that the code size reported for Etos might differ slightly from the code size of these programs when loaded; however, not by much. Table 4 denotes this by a †.

5.5 Erlang vs. other functional languages

We have compared the performance of HiPE with that of several other functional language implementations (Bigloo 2.1c, CML from SML/NJ 110.0.6, CLEAN 1.3.2) on a few small benchmarks (versions of the **qsort**, **fib**, **huff**, and **ring** benchmarks described previously).

On the small sequential benchmarks, CLEAN was consistently the fastest system. The other three systems showed considerable variance in their performance, with no clear winner. Compared to CLEAN, CML was from 1.75 to 4.4 times slower, Bigloo 1.33 to 13 times slower, and HiPE 1.88 to 14.8 times slower. HiPE and Bigloo had comparable performance, except on **qsort** where HiPE was more than twice as fast. On the concurrent **ring** benchmark, HiPE was 1.3 times slower than CML, while the JAM emulator was 1.7 times slower than CML.

See [15] for further details about this comparison.

6. MORE DETAILED MEASUREMENTS

To shed more light on HiPE's performance characteristics compared to the other two Erlang systems from Ericsson, we present some additional measurements on three benchmarks of different sizes: **length**, **Eddie**, and **AXD/SCCT**. These measurements were obtained by enabling the performance instrumentation features of the three systems (see Section 4.6) during their installation; this explains slight mismatches with performance numbers reported in the previous section. A more thorough analysis and comparison of these systems can be found in [14].

length is a small sequential benchmark consisting of two nested loops, one that traverses a list and one that iterates a number of times.

On this benchmark HiPE is 16 times faster than JAM and over 3 times faster than BEAM; see Table 5. The byte-code emulation overhead is evident in this benchmark where HiPE executes only 6 million SPARC instructions while JAM and BEAM execute 49 and 10 million instructions respectively. JAM and BEAM also have problems (10% and 14% of their run time, respectively) with pipeline stalls from branch mispredictions; see Table 7. The same table shows that all three systems have some problems with load stalls (a value is needed before it has been completely loaded into a register) but none of the systems suffers from instruction cache misses.

Eddie is a mildly concurrent (only 5 messages are sent) benchmark which parses HTTP requests. The parser consists of four modules, another two are added for the benchmarking purposes, 4.7% of the calls are to built-in C functions, and some Erlang/OTP standard libraries are heavily used. The benchmark consists of 159 different Erlang functions that are called a total of $\approx 30,000$ times.

On this benchmark, HiPE is 6.3 times faster than JAM and over 2 times faster than BEAM. Here the improved BEAM compiler almost makes up for the emulation overhead: BEAM executes 4 million SPARC instructions which is not much more than HiPE's 3 million and a lot less than JAM's 13 million. All three systems have about the same percentage of pipeline stalls: 36% for JAM and HiPE, and 37% for BEAM.

AXD/SCCT is a time critical part of Ericsson's AXD 301 ATM switch. **SCCT** is responsible for setting up and tearing down connections in the switch. The code we have used is from version 6 of AXD 301, an earlier version than what is used in the product today. This benchmark is more concurrent than **Eddie**, with several processes and over 3000 messages sent. It also uses the built-in functions heavily: about 32% of the execution time for JAM is spent in built-ins. In absolute terms, this time is the same in HiPE; in relative terms, it corresponds to 50% of HiPE's execution time and built-ins start becoming a bottleneck. **SCCT** is a large benchmark: the total size of the 501 called functions is in native code 244,936 bytes (see also Table 4).

	JAM	BEAM	HiPE
length	1	4.5	16.1
Eddie	1	2.6	6.3
AXD/SCCT	1	1.4	1.6

Table 5: Speedups for BEAM and HiPE compared to JAM calculated from the total execution times in clock cycles.

		Mc	*Mi*	*CPI*
length	**JAM**	63	49	1.30
	BEAM	14	10	1.45
	HiPE	4	6	0.70
Eddie	**JAM**	18	13	1.39
	BEAM	7	4	1.54
	HiPE	3	3	1.07
AXD/SCCT	**JAM**	153	101	1.52
	BEAM	111	62	1.79
	HiPE	97	62	1.56

Table 6: Measurements for the benchmarks. The *Mc* column is rounded average execution times in million of cycles. The *Mi* column is the average number of millions executed instructions. The *CPI* column is the number of cycles per instruction (the lower the better).

> On this benchmark HiPE is only about 1.6 times faster than JAM and 1.15 times faster than BEAM. Here the improved BEAM compiler completely makes up for the emulation overhead. This is indicated by the fact that BEAM and HiPE both execute about 62 million SPARC instructions on this benchmark; see Table 6.

The execution times in millions of clock cycles (***Mc***) and millions of executed instructions (***Mi***) for each system and benchmark are shown in Table 6.

The high speedup on **length** for HiPE compared to JAM has two reasons:

1. Because of compilation to native code and various optimizations, HiPE executes less than 12% of the number of SPARC instructions that JAM executes.
2. By utilizing the pipeline better, HiPE executes 1.86 times as many instructions per cycle as JAM.

With fewer instructions executed and better pipeline utilization, HiPE gets a speedup of 16 (8.6*1.86) times over JAM; this is shown in Table 5.

On **Eddie**, as on **length**, HiPE and BEAM execute considerably less instructions than JAM, and HiPE has a lower ***CPI*** than JAM and BEAM. This benchmark uses built-in functions and concurrency. This together with the use of different types of calls probably is the reason that the speedup for HiPE is not as great as it was on **length**. On **Eddie**, all three systems spend more than 35% of the time stalling, mainly on loads. HiPE has significantly higher instruction cache stalls than the other two system, while JAM and BEAM suffer more from branch misprediction stalls; see Table 7.

Because of the large size of the generated native code, on **AXD/SCCT** the absolute number of instruction cache stalls is about twice as many for HiPE (22% of 97 million) as for JAM (6% of 153 million). But taken together, the number of stalls from mispredictions and stalls from instruction cache misses is about the same for JAM and HiPE. All three systems spend about 40% of their total execution time stalling

		%L	*%M*	*%IC*	*%Total*
length	**JAM**	12	10	0	22
	BEAM	9	14	1	23
	HiPE	16	0	2	18
Eddie	**JAM**	23	10	2	36
	BEAM	23	11	3	37
	HiPE	18	4	14	36
AXD/SCCT	**JAM**	23	12	6	41
	BEAM	25	8	7	40
	HiPE	16	5	22	43

Table 7: Percentage of the execution time spent on the most common types of pipeline stalls. The column *%L* shows the percentage of load stalls, *%M* shows the percentage of branch misprediction stalls, *%IC* shows the percentage of instruction cache stalls, and finally the *%Total* column shows the total percentage of stalls. All numbers are based on the number of executed machine cycles.

on this benchmark. This indicates that even though HiPE runs into problems with the instruction cache, HiPE does not suffer more from this than JAM or BEAM suffer from other types of stalls. One reason why HiPE does not achieve the same speedup as on smaller benchmarks is that much of the time is spent in code currently outside HiPE's control, such as built-in functions and garbage collection. Ignoring the time spent in built-in functions, the garbage collector, and the operating system, HiPE is 2 times faster than JAM on this benchmark.

7. CONCLUSIONS AND FUTURE PLANS

This paper has described the architecture and implementation of HiPE, a native code compiler for Erlang. HiPE offers flexible, user-controlled integration between interpreted and native code, is a complete implementation of Erlang, and supports features crucial for telecommunication applications such as concurrency, error handling, and hot-code loading. As our performance evaluation shows, HiPE is the fastest of current Erlang implementations.

HiPE is publicly available as open-source since March 2000. The current release is based on Erlang 47.4.1. Since the Ericsson releases of open-source Erlang will exclusively use the BEAM implementation from now on, we plan to port HiPE to the BEAM run-time system. At present HiPE only runs on one platform, the UltraSPARC. To improve the usefulness of the HiPE system, we are currently developing a code generator for the x86 processor family. We are also developing a new front-end for HiPE so that compilation does not necessarily rely on the outcome of a previous compilation to JAM (or to BEAM). Instead, the new front-end will be based on Core Erlang,[7] an intermediate representation for Erlang developed recently. Since Core Erlang is a fairly high-level functional language, we expect that it should be easier to include optimizations, for example efficient pattern-match compilation, at that level.

Our benchmark results indicate that real-world applications spend a large fraction of their time in built-in standard pro-

[7] See `http://www.csd.uu.se/projects/hipe/corerl`.

cedures. We will therefore investigate this issue further, and extend the scope of HiPE accordingly. This may require improving the compiler, tuning the runtime system, or both.

Acknowledgements

HiPE has greatly benefited from the prior involvement of Christer Jonsson and from various design discussions with Thomas Lindgren, Sven-Olof Nyström, and Richard Carlsson. We also thank some anonymous reviewers for comments which helped improve the presentation of this paper. This research has been supported in part by ASTEC (Advanced Software Technology) competence center.

8. REFERENCES

[1] J. Armstrong. The development of Erlang. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 196–203, June 1997.

[2] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[3] T. P. Baker and G. A. Riccardi. Implementing Ada exceptions. *IEEE Software*, 3(5):42–51, Sept. 1986.

[4] J. R. Bell. Threaded code. *Communications of the ACM*, 16(8):370–373, June 1973.

[5] S. Blau and J. Rooth. AXD 301—A new generation ATM switching system. *Ericsson Review*, 75(1):10–17, 1998.

[6] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Trans. Prog. Lang. Syst.*, 16(3):428–455, May 1994.

[7] R. K. Dybvig. *Three Implementation Models for Scheme*. PhD thesis, Department of Computer Science, University of North Carolina at Chapel Hill, 1987. Technical Report TR87-011. Available from: http://www.cs.indiana.edu/scheme-repository/.

[8] M. Feeley and M. Larose. Compiling Erlang to Scheme. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Principles of Declarative Programming*, number 1490 in LNCS, pages 300–317. Springer-Verlag, Sept. 1998.

[9] A. D. Gordon. How to breed hybrid compilers/interpreters. Technical Report ECS-LFCS-88-50, Department of Computer Science, University of Edinburgh, 1988.

[10] D. Gudeman. Representing type information in dynamically typed languages. Technical Report TR 93-27, University of Arizona, Department of Computer Science, Oct. 1993.

[11] P. H. Hartel et al. Benchmarking implementations of functional languages with "pseudoknot", a float intensive program. *Journal of Functional Programming*, 6(4):621–655, July 1996.

[12] B. Hausman. Turbo Erlang: Approaching the speed of C. In E. Tick and G. Succi, editors, *Implementations of Logic Programming Systems*, pages 119–135. Kluwer Academic Publishers, 1994.

[13] R. Hieb, R. K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 66–77, June 1990.

[14] E. Johansson, S.-O. Nyström, T. Lindgren, and C. Jonsson. Evaluation of HiPE, an Erlang native code compiler. Technical Report 99/03, ASTEC, Uppsala University, 1999.

[15] E. Johansson, M. Pettersson, K. Sagonas, and T. Lindgren. The development of the HiPE system: Design and experience report. Technical report, ASTEC, Uppsala University, 2000. In preparation.

[16] R. E. Jones and R. Lins. *Garbage Collection: Algorithms for automatic memory management*. John Wiley & Sons, 1996.

[17] P. Klint. Interpretation techniques. *Software – Practice and Experience*, 11(9):963–973, Sept. 1981.

[18] G. Lal. MLRISC: Customizable and reusable code generators. Unpublished technical report available from: http://www.cs.bell-labs.com/~george, 1996.

[19] S. Peyton Jones, N. Ramsey, and F. Reig. C--: A portable assembly language that supports garbage collection. In G. Nadathur, editor, *Principles and Practice of Declarative Programming: Proceedings of International Conference PPDP'99*, number 1702 in LNCS, pages 1–28. Springer-Verlag, Sept. 1999.

[20] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Computer Science. Prentice-Hall, 1987.

[21] R. M. Stallman. Using and porting gcc. Technical report, The Free Software Foundation, 1993.

[22] Sun Microsystems. UltraSPARC™ User's Manual. Technical report, Sun Microelectronics, Palo Alto, CA, 1997.

[23] S. Torstendahl. Open telecom platform. *Ericsson Review*, 75(1):14–17, 1997. See also: http://www.erlang.se.

[24] R. Virding. A garbage collector for the concurrent real-time language Erlang. In H. G. Baker, editor, *Proceedings of IWMM'95: International Workshop on Memory Management*, number 986 in LNCS, pages 343–354. Springer-Verlag, Sept. 1995.

[25] P. Wadler. An angry half-dozen. *SIGPLAN Notices*, 33(2):25–30, Feb. 1998.

[26] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.