HiPE: High Performance Erlang

Erik Johansson
Sven-Olof Nyström
Mikael Pettersson
Konstantinos Sagonas

Technical Report
October 1999
ASTEC 99/04

HiPE: High Performance Erlang

Erik Johansson, Sven-Olof Nyström,
Mikael Pettersson, and Konstantinos Sagonas

Computing Science Department
Uppsala University, Sweden
e-mail: {happi,svenolof,mikpe,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, a 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 a group at Ericsson to address the needs of large-scale distributed soft real-time control applications [3]. 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; such as scheduling of light-weight 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, it is generally acknowledged that Erlang is among the "success-stories" of declarative programming languages; see e.g. Wadler's article [21]. Also, the experience reported in [2, 6] is that Erlang allows telecommunication systems to be programmed with less effort and fewer errors than by using conventional programming language technology. 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 heavily rely 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 [2]: '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 worse than that of good implementations of other functional programming languages; see also [8, 9]. 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 [6]. 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 by, say even 20%, would be more than welcome by the AXD 301 engineering team, let alone Ericsson. The reason: this 20% directly corresponds to the ATM switch being capable of servicing around 20% more requests, which translates to higher revenues for Ericsson as it puts AXD 301 way ahead of its competitors' products!

Currently, complete implementations of Erlang are 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) use a sufficiently low-level and fast language such as C or 2) the recently proposed `C--` [16] as a portable assembly language, 3) use a retargetable code generator such as ML-RISC [15] or 4) the `gcc` back-end [18], 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.

This paper describes 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. More specifically, HiPE currently uses the JAM emulator [2] 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 describing HiPE, we discuss technical issues that this emulated/native code integration entails and how we dealt with them. We pay special attention to supporting *hot-code loading* (see next section) in HiPE, and apply various standard compiler optimizations in our native code generation. The end result of our effort currently appears to justify its name!

The rest of the paper is organized as follows: 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. Some aspects of HiPE's instrumentation and support for profiling are discussed in Section 5. Section 6 compares and analyses the performance of HiPE against all other existing Erlang compilers on both "standard" small benchmarks and on large programs from real-world applications of Erlang. We end this paper with some concluding remarks.

2 The Erlang Language: A brief introduction

Erlang[1] 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, the first occurrence of a variable is its binding instance, and 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. 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 fails.

An Erlang module defines a number of functions. Only explicitly exported functions may be called from other modules. Calls to functions in different modules, *remote calls*, must give the name of the module of the called function. During execution of functions, last call optimization is performed. Memory management in Erlang is automatic through garbage collection (as in all other functional programming languages), but the real-time concerns of the language call for bounded-time garbage collection techniques (see [20]).

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 as small as possible, but comes with a relatively big set of libraries containing *built-in functions* (known as *BIFs*). With the Open Telecom Platform (OTP) middleware, Erlang is further extended with a library of standard solutions to common requirements of

[1] Named after the Danish mathematician Agner Krarup Erlang (1878–1929).

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 exists as a new component (currently 30,000 lines of Erlang code for the compiler and 3,000 lines of C and assembly code in the runtime system) added to an otherwise mostly unchanged JAM system; only the JAM emulator and the memory management subsystem have been extended to be aware of HiPE. Because of this tight integration, we describe relevant aspects of the basic JAM system here; Section 4 continues with HiPE-specific implementation details.

Until recently, Erlang was only available within the Ericsson company and to a few research groups outside Ericsson. The first version of HiPE, HiPE 0.28 [12], was based on Erlang 4.3.1, and the current version, HiPE 0.90, is based on Erlang 4.5.3. Being based on proprietary code, these versions of HiPE are not publicly available. Erlang has recently been made available as Open Source. We are currently porting HiPE 0.90 to Open Source Erlang 4.7.4.1, and expect to have a public release of HiPE 1.0 before the end of 1999.

3.1 The JAM System

The standard Ericsson implementation of Erlang is based on the JAM virtual machine. The JAM is a virtual stack machine whose primitive operations closely correspond to the Erlang language.

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

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.

Remote (non-local) function calls are represented by triples (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, both the JAM opcode and its parameters at this call site are 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.

The JAM emulator. The JAM emulator is a single C function which executes JAM instructions represented as bytecodes. It 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 Fig. 1. At return, `sp` is reset to `ap`, `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` →	...

Fig. 1. JAM stack on entry to a function

Tailcalls 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 will have to be relocated whenever there is a tailcall and the caller and callee have different number of parameters. At a tailcall, 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 [4]. 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 bytecode 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

---

[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 tailcalls [7, Section 4.6.1].

record containing the current catch frame is found. The unwinding process also restores the `sp`, `fp`, `ap`, and `cc` registers.

3.2 Processes and Memory Management

The JAM implementation uses a hybrid memory management system which combines several techniques.

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.

Each Erlang process has a PCB, a stack, and a private heap for the data structures it creates. When an Erlang process terminates, its memory resources are immediately deallocated and made available for reuse. The idea behind this is that each process is expected to have only a small amount of live data, so garbage collecting a single process is expected to be a fast operation. (A standard generational stop-and-copy collector is used.) Also, there is no delay from the point when a process terminates to the point when its memory can be reused. The disadvantage is that the garbage collector cannot handle references from one process' heap to another's; therefore, message passing always implies copying. Messages are expected to be small, however.

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

A global database, the Erlang Term Storage tables, is accessible from all Erlang processes within a node. It is implemented essentially as a an additional but specialised process, so storing and retrieving data implies copying, like when sending messages.

Erlang `binary` objects are immutable sequences of binary data, and are often relatively large. A `binary` is represented by a data area, and a small header which contains a pointer into the data area. When a binary is split into sub-binaries (a frequent operation), new headers are created but the data area is not copied. Binaries are allocated in memory visible to all Erlang processes in a node; when sent in a message, a binary is passed by reference. Although the standard garbage collector is based on copying, binaries are mark-sweep collected.

3.3 The Complete Erlang System

The complete system consists of the runtime system and the Erlang libraries. The runtime system contains the JAM emulator, the garbage collector, the process scheduler, the standard procedures which are implemented as C functions, the OS interface, and other bits and pieces. The Erlang libraries contain the Erlang compiler, support for input/output, networking, and building client-server applications, and the Open Telecom Platform (OTP) libraries.

4 HiPE – System Overview

The HiPE compiler is called as an ordinary Erlang function, within a running Erlang system. Given the name of an existing Erlang function, it translates that function's JAM bytecodes to SPARC V9 machine code, and then updates the symbol table so that future calls invoke the native code.

HiPE compiles a single Erlang function at a time. As mentioned before, telecom applications tend to be very large, and code size is a real concern. Code expansion during compilation is in the order of a factor 10–20, so it would be inappropriate to compile all JAM functions to machine code. Instead, the programmer is expected to identify (usually using profiling tools) those functions and call paths that would benefit most from compilation to native code.

We want native code to execute as fast as possible, even if that means complicating the mode-switching interface between native and emulated code.

Historically, Erlang libraries were only present as JAM bytecode files. Therefore, HiPE is designed to take already-loaded JAM bytecode instead of Erlang source code as input, and it only stores the generated machine in memory, not in files[3]. In this respect, HiPE resembles a JIT compiler. However, compilation always occurs as a side-effect of an explicit call to the HiPE compiler.

Code compiled by HiPE uses the same runtime system and the same built-in C functions as the JAM emulator. It can, and is often used to, recompile standard Erlang libraries into more efficient native code.

4.1 The HiPE Compiler

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

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. A register renaming post-pass ensures that every register has a single definition[4].

[3] This also means that HiPE may lose some performance since the JAM compiler is not aggresively optimising. A future version of HiPE will use a new compiler front-end currently under development at Uppsala University and Ericsson.

[4] This is not a semantic requirement, but it helps later compilation passes.

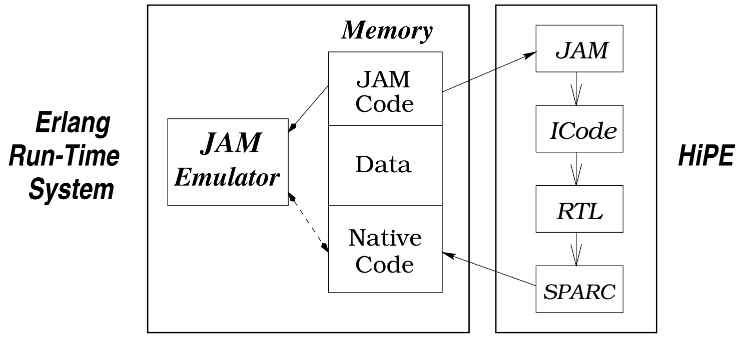


Fig. 2. Intermediate representations in HiPE.

The ICode is then optimised with 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 same optimisations that were performed on ICode are then applied 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 simple graph-colouring register allocator. 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 already executing in 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 areas. The exported functions always refer to the *current* code segment. At a remote function call, `module:function(params...)`, 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.

HiPE uses *code patching* to eliminate these dynamic lookups from native code. In native code, each function call is implemented as a subroutine call to an absolute address. When the caller's code is being linked, the linker initialises the call to directly invoke the callee's code. 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. The linker keeps track of all call sites, and dynamically upgrades them as modules are loaded and as functions are compiled to native code.

Even though it is possible to use code patching to reduce the overhead incurred on remote procedure calls by hot-code loading, hot-code loading still incurs runtime costs and makes optimisations, such as inlining, harder.

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 to instead invoke the JAM emulator.

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 does. However, using the same calling convention in native code as in JAM is not a good idea, 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, and uses only a single-word stack frame for preserving the previous return address.

HiPE uses two stacks for each process, one for emulated code (the *estack*) and one for native code (the *nstack*). An early version of HiPE [12] used only one stack, but that was complex and difficult to implement correctly. Our current dual-stack approach is complicated by the fact that each catch-frame contains a link to the previous catch-frame, which may be on the other stack, and either stack may have to be relocated if a process needs more stack space. The dual-stack discipline also complicates the implementation of tail-recursive calls between native and emulated code, as described below.

4.4 Switching from emulated to native mode

Each JAM function is prefixed with a header, which records the location of that function's native code, if it has been compiled. At call sites, the JAM emulator checks this header to determine if a switch to native code should be done. At return sites and during the search for an exception handler, a check is made if the return address has a magic value; if so, a switch to native code is done.

The mode switch interface, a C function, switches from emulated to native mode, calls the native code, and on return switches back to emulated mode. The exact actions vary, depending both on the *kind* of mode switch (call, tailcall, return, exception, suspend, wait) and its direction.

Each of the two stacks is to be viewed as a sequence of segments. When there is a recursive call from one mode to the other, there will be a segment boundary at the top of the caller's stack. The "current" segment is the portion of the stack between the stack pointer and the most recent segment boundary.

When calling a native code function from emulated code, the first five parameters are copied to the PCB, and then loaded into registers by a small assembly-coded stub. Parameters beyond the fifth are pushed onto the nstack.

The interface uses *magic* frames. These frames are formatted as normal call or catch frames, but their return addresses cause control to flow back to the mode-switch interface.

call: Emulated code has pushed a call frame on the estack and now tries to invoke a native code function.

1. push magic call frame on estack
2. push magic catch frame on nstack
3. copy parameters from estack to PCB and nstack
4. call `native_interface()` to invoke the code

tailcall: Emulated code is performing a tailcall. The caller's call frame is on the estack. If this is a magic frame, then native code called emulated code, which now tailcalls native code.

1. if the top estack call frame is not magic, then perform the same actions as for a call; otherwise:
2. copy parameters from estack to PCB and nstack
3. pop magic call and catch frames from estack
4. call `enter_native()` to invoke the code

return: Native code called emulated code, which is now returning.

1. move (copy and pop) return value from estack to PCB
2. pop magic catch frame from estack
3. call `ret_to_native()` to invoke the code

exception: Native code called emulated code, pushing a magic catch frame. An emulated-mode exception invoked this catch frame, and the exception is now re-thrown on the nstack.

1. move exception value from estack to PCB
2. pop nstack to the position of the current exception handler
3. call `ret_to_native()` to invoke the code

resume: A native-mode process had suspended, and is now awakened.

1. set resume cause flag in PCB
2. call `ret_to_native()` to invoke the code

4.5 Switching from native to emulated mode

The switch back from native to emulated mode is triggered when the process must suspend, when it invokes a native-to-emulated call stub inserted by the linker, or when it invokes a magic return or exception handler address provided by the mode-switch interface.

return: Emulated code called native code, which is now returning. The current nstack segment is empty, except for a single magic catch frame. The estack has a magic and a real call frame on top.

1. pop magic catch frame off nstack

2. pop magic and real call frames off estack
3. move return value from PCB to estack
4. return to JAM emulator with status **continue**

call: Native code calls or tailcalls emulated code.

1. push magic catch frame on estack[5]
2. move parameters from PCB and nstack to estack
3. push magic call frame on estack
4. return to JAM emulator with status **continue**

exception: Emulated code called native code, which is now throwing an exception. The current nstack segment is empty. The estack has a magic and a real call frame on top.

1. pop magic and real call frames off estack
2. move exception value from PCB to estack
3. return to JAM emulator with status **exception**

suspend: The process' time-slice has run out.

1. mark PCB as suspended but runnable
2. return to JAM emulator with status **suspended**

wait: The native code suspends due to an empty message queue.

1. mark PCB as suspended waiting for message
2. return to JAM emulator with status **suspended**

4.6 Possible Improvements of Emulated code

To improve performance of emulated code, we could extend the HiPE linker to also record all call sites in JAM bytecode, and to patch those when their targets are compiled to native code. This could also be used to replace remote calls using dynamic lookup with direct calls, and to automatically update call sites if the target module's code is upgraded.

5 Instrumentation of HiPE

We have enhanced HiPE's run-time system with performance instrumentation features that can be selectively included or excluded when the run-time system is built. These instrumentation features come in two forms:

software counters: These counters keep track of how often various operations of interest are performed. For example, counters keep track of the number of times each Erlang function is called, either locally, remotely, or through a meta-call (`apply`). They can also count calls to built-in functions, how many times each JAM instruction is executed, and how many times control is passed between emulated and native code.

[5] A recursive call from emulated to native code, followed by a tailcall from native to emulated code, will leave a magic catch frame on the nstack. This frame will be deallocated when the emulated code returns or throws an exception.

performance instrumentation counters (PICs): The PICs are based on the Sun UltraSPARC's performance instrumentation facilities [19]. PICs are made accessible to the user through a builtin function, and they are typically used to measure how much time is spent in a region of code, and to give 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. Currently, HiPE uses PICs to measure time spent in garbage collection, each built-in-function, native code, and each time-slice. The instrumentation counts both elapsed cycles and issued instructions, making it possible to determine the CPI (cycles per instruction) ratio.

For more details on the instrumentation, the reader is referred to [13].

6 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. 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 4.7.3. The version of Etos used is the latest one, 2.3[6]. HiPE and JAM have been described before; we discuss BEAM and Etos:

BEAM [10] is an emulator for an abstract register-based machine, influenced by the Warren Abstract Machine (WAM) [1] used in many Prolog implementations. Compared to JAM, the translation to the abstract machine is more advanced. For example, the treatment of pattern matching is better in the BEAM, even though a full pattern matching compiler (like that in e.g. [17]) is not implemented. Also, BEAM uses a direct-threaded emulator [5, 14] using `gcc`'s labels as first-class objects extension [18]: instructions in the abstract machine are addresses to the part of the emulator that implement the instruction.

Etos [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 [11]. When a process is suspended, the stack is "frozen" so that no frame currently on the stack can be deallocated. Thus, the stack of a suspended process will occupy a portion of the stack. When control returns to

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

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.

We start our performance comparison using the following set of "standard" small sequential benchmarks:

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.
tak Takeuchi function, uses recursion and integer arithmetic intensely. Calculates `tak(18,12,6)` 1000 times.
smith The Smith-Waterman DNA sequence matching algorithm. Matches one sequence against 100 others; all of length 32. This is done 30 times.

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

decode Part of a telecommunications protocol. Decodes an incoming `binary` message 500,000 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
tak	13.5	12.8	140.1	100.2
smith	11.4	11.6	114.6	53.9
decode	22.8	52.4	67.8	49.0

Table 1. Execution times (in seconds) for small sequential benchmarks.

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 Fig. 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, 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.

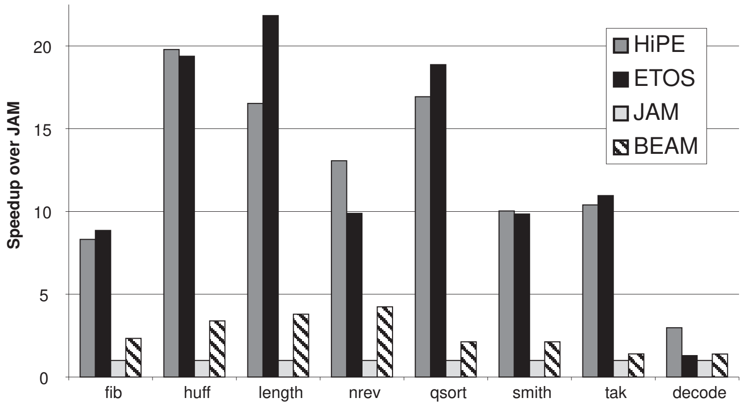


Fig. 3. Performance speedup compared to JAM for small benchmarks.

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.

stable Solves the stable marriage problem for 10 men and 10 women 5,000 times.

life Executes 1,000 generations in Conway's game of life on a 10 by 10 board where each square is implemented as a process.

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. Execution times (in seconds) for small concurrent benchmarks.

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 Fig. 4(a). 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.

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.

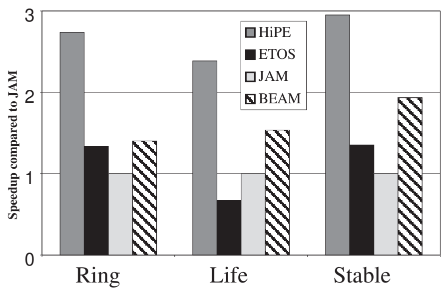
Eddie An HTTP parser which handles 30 complex http-get requests. Excluding the OTP libraries used, it consists of 6 modules for a total of 1,882 lines of Erlang code. The benchmark is executed 1,000 times.

AXD 301/SCCT This is the time-critical software part of the AXD 301 switch mentioned in the introduction. Excluding standard libraries, it consists of 73 modules of $\approx 95,000$ lines of Erlang code. This actual benchmark of the ATM switch sets up and tears down a number of connections 100 times.

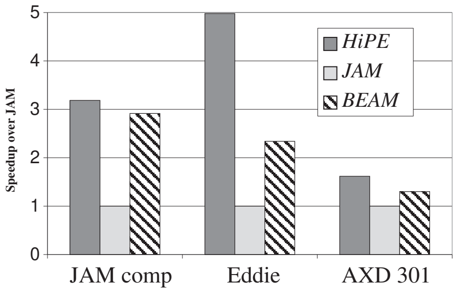
Benchmark	HiPE	JAM	BEAM
JAM Compiler	5.4	17.2	5.9
Eddie	18.8	93.6	40.0
AXD 301/SCCT	68.0	109.9	84.5

Table 3. Execution times (in seconds) for large benchmarks.

Table 3 shows the results of this comparison [7]. HiPE is once again the fastest system, but as benchmarks get larger, programs tend to spend more and more of their execution time in built-ins from the Erlang standard library (e.g. the **AXD 301/SCCT** benchmark extensively uses the built-ins to access the shared database on top of Erlang term storage; see [13]), so its speedup compared to BEAM drops. Still, HiPE is 25% faster than BEAM on the largest benchmark, and considerably faster than JAM; see also Fig. 4(b).



(a) Concurrent



(b) Large

Fig. 4. Performance speedup compared to JAM for concurrent & large benchmarks.

7 Concluding Remarks and Future Plans

HiPE is a native code compiler for Erlang. It offers flexible integration between interpreted and native code, and supports features crucial for telecommunication applications such as concurrency, error handling, and hot-code loading. As our performance evaluation shows, it is the fastest of current Erlang implementations.

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

We plan to release a version of HiPE, based on open-source Erlang 4.7.4.1, before the end of 1999. Since future versions of open-source Erlang will use the BEAM implementation, 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 plan to develop a code generator for the x86 processor family. We are also developing a new front-end for HiPE that will not rely on the JAM. Instead, the new front-end will be based on Core Erlang[8], 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-matching compilation, at that level.

References

1. H. Ait-Kaci. *Warren's Abstract Machine*. The MIT Press, Cambridge, MA, 1991.
2. J. Armstrong. The development of Erlang. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 196–203, June 1997.
3. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.
4. T. P. Baker and G. A. Riccardi. Implementing Ada exceptions. *IEEE Software*, 3(5):42–51, Sept. 1986.
5. J. R. Bell. Threaded code. *Communications of the ACM*, 16(8):370–373, June 1973.
6. S. Blau and J. Rooth. AXD 301—A new generation ATM switching system. *Ericsson Review*, 75(1):10–17, 1998.
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 at the Internet Scheme Repository, 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. P. H. Hartel et al. Benchmarking imlementations of functional languages with "pseudoknot", a float intensive program. *Journal of Functional Programming*, 6(4):621–655, July 1996.
10. 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.
11. 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.
12. E. Johansson and C. Jonsson. Native code compilation for Erlang. Uppsala master thesis in computer science 100, Uppsala University, 1996.
13. E. Johansson, S.-O. Nyström, T. Lindgren, and C. Jonsson. Evaluation of HiPE, an Erlang native code compiler. Technical Report 99/03, ASTEC, Uppsala, 1999.
14. P. Klint. Interpretation techniques. *Software – Practice and Experience*, 11(9):963–973, Sept. 1981.

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

15. G. Lal. MLRISC: Customizable and reusable code generators. Unpublished report available from: http://www.cs.bell-labs.com/~george., 1996.
16. 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.
17. S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Computer Science. Prentice-Hall, 1987.
18. R. M. Stallman. Using and porting gcc. Technical report, The Free Software Foundation, 1993.
19. Sun Microsystems. UltraSPARC™ User's Manual. Technical report, Sun Microelectronics, Palo Alto, CA, 1997.
20. 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.
21. P. Wadler. An angry half-dozen. *SIGPLAN Notices*, 33(2):25–30, Feb. 1998.