

The Development of the HiPE System: Design and Experience Report

Erik Johansson¹, Mikael Pettersson¹, Konstantinos Sagonas¹, Thomas Lindgren²

¹ Computing Science Department, Uppsala University, Sweden. e-mail: {happi,mikpe,kostis}@csd.uu.se

² Bluetail AB, Sweden. e-mail: thomasl@bluetail.com

Abstract. The concurrent functional programming language Erlang has been designed to ease the development of large-scale distributed soft real-time control applications. So far, it has been used quite successfully in industry, both within Ericsson Telecom, where it was designed and developed, and by other companies. This “declarative language success-story” has taken place despite the fact that Erlang implementations are slow compared with implementations of other functional languages.

Wanting to improve the performance aspects of publicly available Erlang implementations, which are based on emulators, we embarked on a project called HiPE (High-Performance Erlang) whose aim has been to develop an efficient just-in-time native code compiler for Erlang (called the HiPE system). Since its start in 1996, the system has gone through various (re-)design phases, partly due to implementation choices that did not turn out to be as promising as they appeared on paper, but mainly due to changes in Ericsson’s Erlang system upon which the HiPE system is built.

In this article, we describe how the HiPE system was developed, what it currently looks like and its current performance. We critically examine design decisions that we took, and the main lessons learnt from implementing them. Finally, we also report on our experiences from trying to keep up with the concurrent development of Ericsson’s base Erlang system. As such, this article both documents the HiPE system and can serve as possible guidance to anyone wishing to attempt a similar feat.

Keywords: Programming Language Implementation – Concurrent Programming – Functional Programming – Erlang – Virtual Machines and Compilation Methods – Hacking

Correspondence to: Erik Johansson.

1 Introduction

Erlang is a strict, dynamically typed functional programming language with support for concurrency, communication, distribution, fault-tolerance, on-the-fly code reloading, automatic memory management, and multiple platforms [2]. It has been designed with the aim of easing the programming of large soft real-time control systems which are commonly developed by the telecommunications industry. Judging from commercial applications written in Erlang, often consisting of upwards of half a million lines of code, the language currently seems to be quite successful in its domain. This success story has taken place despite the fact that implementations of Erlang are based on virtual machine emulators and are thus relatively slow, even compared with implementations of other functional languages. In this respect, Erlang bears a striking resemblance to another modern phenomenon, Java, which also achieved its initial success without high-performance implementations with faster compilers coming along later.

In spite of Erlang’s success so far, the need for speed is sometimes pressing in the competitive telecommunications market. Availability of fast implementations, although by no means a necessary condition for a programming language’s success, helps in maintaining interest in the language, sustains its continued development, and aids in the language’s adoption by new projects in this industrial setting. Furthermore, developing a high-performance compiler for an industrially relevant language provides an excellent opportunity to transfer technology between academia and industry and evaluate the effect of new optimization techniques on real-world applications rather than on toy programs.

Prompted by these reasons, we have embarked on an ASTEC (Advanced Software Technology) project, partly funded by Ericsson Development AB, the goal of which is to develop a high-performance implementation of Er-

lang. Our approach to the efficient execution of Erlang programs has been to develop a just-in-time native code compiler that can be used as an add-on component to an otherwise mostly unchanged Open Source Erlang implementation. The resulting system, called HiPE (High-Performance Erlang), allows its user to selectively compile single functions or whole modules into native code, thus combining the performance characteristics of a native code compiler with the benefits of an emulator implementation.

The purpose of this article is to fully describe HiPE: its design and development process since the beginning of the project (Section 3), its current architecture and aspects of its implementation that might be relevant or applicable to other similar systems (Section 4), and its performance compared with current implementations of Erlang and other functional languages (Section 6). In addition to documenting our approach, we use this opportunity to critically examine the design choices we made (Section 5). We hope that our experience and the issues discussed are of interest to other programming language implementors or of use to those that want to engage themselves in similar projects. We begin with a brief review of Erlang’s characteristics.

2 Erlang: The Language and its Applications

As mentioned, Erlang¹ is a dynamically typed, strict, concurrent functional language. Erlang does have function closures, but typical Erlang programs are mostly first-order. Erlang’s basic data types are atoms, numbers (floats and arbitrary precision integers), process identifiers, and references; compound data types are lists and tuples. There is no destructive assignment of variables or data, and the first occurrence of a variable is its binding instance. Function rule selection is done with pattern matching. Erlang 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 large, 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 through 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 provides mechanisms for allowing a process to timeout while waiting for messages and a catch/throw-style exception mechanism for error handling.

For programming in-the-large, Erlang comes with a module system. An Erlang module defines a number of functions. Only explicitly exported functions may be called from other modules. Calls to functions in different modules, called *remote calls*, are done by supplying the name of the module of the called function. Tail call optimization is a required feature of Erlang. As in other functional languages, memory management in Erlang is automatic through garbage collection. The real-time concerns of the language call for bounded-time garbage collection techniques; see [30,18]. In practice, garbage collection times are usually small as most processes are short-lived or small in size.

Erlang is used in “five nines” high-availability (i.e., 99.999% of the time available) systems, where downtime is required to be less than five minutes per year. Such systems cannot be taken down, upgraded, and restarted when software patches and upgrades arrive, since that would not respect the availability requirement.

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). Once the old code is no longer in use, the old module is unloaded.

The Erlang language was purposely designed to be small, but it comes with libraries containing a large set of *built-in functions* (known as *BIFs*). With the Open Telecom Platform (OTP) middleware [29], Erlang is further extended with a library of standard solutions to common requirements in telecommunication applications (real-time databases, servers, state machines, process monitors, load balancing), standard interfaces (CORBA), and standard communication protocols (e.g., HTTP, FTP).

Erlang is currently used industrially both by Ericsson Telecom and by other companies for the development of high-availability servers and networking equipment. Some example products built using Erlang/OTP are: AXD/301, a scalable ATM switching system [4], ANx, an ADSL delivery system [22], a switching hardware control system, a next-generation call center, and a suite of scalable internet servers from Bluetail AB. Since 1994, the annual Erlang User Conference is the principal forum for reporting work done in Erlang and provides a record of Erlang’s evolving industrial use; additional information about Erlang applications can be obtained through the relevant pages at www.erlang.org.

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

3 An Account of HiPE's Development

In this section we will describe the history of HiPE, while briefly addressing some implementation details and the rationale behind some design decisions we took. We divide the description into four parts, corresponding to the four major revisions of the HiPE system:

1. A first attempt, written in C, gave some insight on how to address the problem of efficiently implementing Erlang and showed that considerable speedup could be achieved using relatively simple methods.
2. A flexible and more easily extendible compiler design, mostly written in Erlang, made it possible to experiment with different optimization techniques and measure their impact on some “real-world” applications of Erlang.
3. An Open Source Erlang distribution from Ericsson made it possible for HiPE to be publicly released, get some users and input from the outside world.
4. A strong coupling of the HiPE compiler with Ericsson's Erlang/OTP system is underway. The goal is to have HiPE as a standard component in future releases of Open Source Erlang.

3.1 JERICO: The first prototype

The starting point of the HiPE system was a Master's thesis project in the summer of 1996 [15]. The goal was to develop an optimizing compiler, called *JERICO*, that would substantially improve the performance of Erlang programs.

One approach that was briefly considered was to use the Java Virtual Machine (JVM [20]) as a back-end—this was at the time when Java was just starting to become a popular language. It was soon realized that the architecture of the JVM is not well-suited for a dynamically typed language such as Erlang. The JVM provides no support for tagged data items, so for example integers have to be wrapped, and it is awkward to get proper tail-recursion which is crucial for Erlang programs. In addition, compiling to JVM implies losing control over the efficiency of light-weight threads; a feature critical for the performance of typical Erlang applications; see also [12] which compares the performance of Erlang processes and Java threads. Consequently, the idea to compile to JVM was quickly abandoned and instead we decided to aim for a direct compilation to native code. The chosen architecture was SPARC V8; according to Ericsson this was the most common general purpose platform for Erlang applications at the time.

Since we wanted to develop a compiler that worked for the whole Erlang language and not just a toy compiler for a subset of Erlang, we decided to base our compiler on the stable and working Erlang runtime system made by Ericsson.

At this time there were two Erlang systems concurrently being developed at Ericsson:

JAM The older system with a stack-based abstract machine.

BEAM A relatively new system based on a register abstract machine, influenced by the Warren Abstract Machine (WAM) [31] used in many Prolog implementations. At that time, the BEAM system had an option to compile Erlang programs to native code via C [13]; this option was not very robust and was later removed.

Both systems used the same runtime system and used similar data representations [11]. The BEAM system then was quite complex and not really stable. Also, at that time, BEAM had not proven itself substantially faster than JAM. The JAM system on the other hand was quite stable and significantly simpler; it was byte-code interpreted and had less than 256 instructions. We decided that this would be a good starting point for our compiler: we could translate the generated JAM byte-code into an internal intermediate representation and then optimize it before generating native code.

The source code for JAM version 4.3 (written in C) was provided to us by Ericsson after we signed a non-disclosure agreement. In order to get tight integration between our compiler and the runtime system we chose to implement our compiler entirely in C.

3.1.1 JERICO: Implementation issues

Compiler In the Ericsson implementations of Erlang, the smallest unit of compilation is a module, but we decided early on that the user of our system should be able to choose to selectively compile a single—presumably time-critical—function at a time to native code. This way, users could combine the compact representation of emulated byte-code with the efficiency of (usually larger) native code. This feature in HiPE is potentially very important for large telecom applications, where typically only a small portion of the code is time-critical while the remaining code deals with error correction and maintenance.

In order to help users decide which functions to compile to native code, the HiPE system provides some simple profiling tools. One of these measures the number of times an emulated function is called. Since recursion is the only way to implement a loop in Erlang, this means that simply counting the number of calls to functions will quickly identify program *hot-spots* (i.e., program points where most of the execution time is spent) which can then be compiled to native code.

Based on this feature, it is easy to build a (just-in-time) hot-spot compiler for Erlang. Indeed, in JERICO one could set a trigger level on the number of calls to emulated functions before these would automatically be compiled to native code. Since the compiler was implemented in C and integrated in the runtime environment,

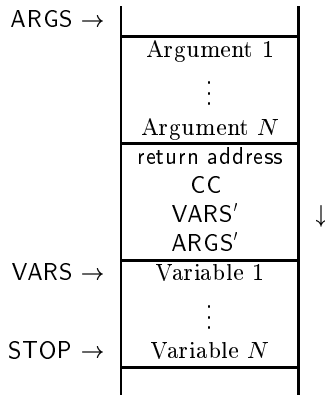


Fig. 1. A JAM stack frame

this meant that execution of Erlang code would be temporarily interrupted while compilation took place. We did no thorough measurements on the performance impact of this feature, but we often enabled it while using the interactive Erlang shell, and we did not notice any degradation in system responsiveness.

The translation from JAM code to the compiler’s intermediate three-address code was done in a straightforward way and left some opportunities for optimization. For example, since JAM was a stack machine there would be a push each time a variable was referenced. This push would be translated to a register copy which would often be unnecessary. In order to improve code quality, the JERICO compiler performed constant propagation, constant folding, unreachable code elimination, and dead code removal [1,21]. The JERICO compiler also implemented a simple delay slot filler which only looked in the basic block preceding the branch for suitable instructions. Register allocation was based on a simple graph coloring algorithm; spilling was not implemented. If the register interference graph could not be colored, or a function’s control flow graph was large and data flow analysis would take too long, the function would remain emulated.

Calling conventions and stack frames The JAM instruction set is simple and the instructions have no knowledge of the current frame size. Several pointers are instead used to keep track of local variables, arguments, and the stack top. These pointers are kept in the VARS, ARGs, and STOP JAM registers. When executing a function call, only ARGs and VARS need to be saved since ARGs always points to the same place as the caller’s STOP.

In order to provide descriptive error messages, the JAM emulator also maintains a current-call register, CC, which always points to the start of the current function’s byte-codes. CC is set at entry to a function, and saved before recursive function calls. This leads to a JAM stack frame with four words in addition to any local variables and arguments; see Fig. 1. In these four words, the return address points to the byte-code instruction to return

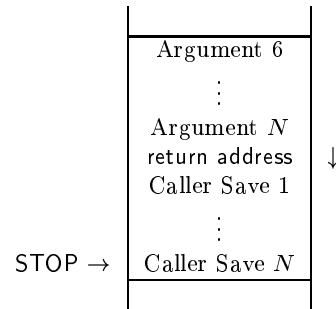


Fig. 2. A native stack frame

to, CC belongs to the caller, and VARs’ and ARGs’ are pointers to the caller’s local variables and arguments.

All the values on the stack are tagged, including the return address. In JAM version 4.3.1, the tags were in the 4 most significant bits of the word and the tag for “FRAME” was 4 zeroes. As long as the system allocated all code below the 2^{28} boundary, each return address would automatically be tagged as a FRAME.

In native code, the JERICO compiler passed the first five arguments in real machine registers. Apart from local variables, only the return address was saved on the stack. Since the compiler knew the format of the stack frame, the ARGs and VARS registers were not needed. Furthermore, the CC register was not needed either since the current function could be identified via the program counter. The format of a native stack frame is shown in Fig. 2.

The JERICO runtime system used the same memory area for the native and the JAM stacks, stacking native frames and JAM frames (see Fig. 2 and 1) on top of each other. Small “dummy” frames were placed between frames of different types to indicate a transition between emulated and native code. Most bugs we encountered originated from the emulated/native-code integration and the rather hairy stack we ended up with. This scheme was later abandoned; see also Section 3.2.

Backpatching In order to facilitate *hot-code loading* and recompilation in an interactive system we implemented a scheme where call sites were patched when the target of the call was updated. For each function, we kept a list of all its callers, and another list of all call sites in the function and their destinations. When a function was recompiled, all callers were updated (backpatched) to call the new function, and references to the old function were removed from all functions it called.

3.1.2 Performance of the JERICO compiler

The JERICO compiler performed quite well on small benchmark programs. It was frequently attaining a factor of 10 speedup over JAM [15] and was slightly faster than BEAM even when BEAM was generating native code using *gcc*. On the other hand, JERICO had prob-

lems with scaling up to compile large systems (e.g., tens of thousands of lines of code) and it was difficult to develop and debug new optimizations rapidly in it.

By examining the emitted code, we also noticed the importance of low-cost procedure calls and optimized those by shrinking the stack frame size from four extra words to one.

Another point confirmed by our measurements was that, even in concurrent applications, most time is spent running sequential code (“in between process communications”). The same measurements also indicated that process communication would start being a bottleneck for those applications, only when the system became 2–3 times faster. We therefore decided to concentrate on sequential optimizations.

3.2 The HiPE system before Open Source Erlang

With the Master’s thesis finished and the JERICO compiler’s performance evaluated on small programs, the natural next step was to make the system more robust, add more code optimizations to the compiler, and evaluate its performance on industrial applications instead of small benchmarks. It was soon realized that—especially in the context of an academic project with limited manpower—the optimizations we wanted to add would be much easier to develop in Erlang than in C. To get a more flexible system that would allow us to easily add new optimizations we decided to rewrite the compiler from scratch in Erlang. At the same time, HiPE was coined as the name of the project (and the system).

3.2.1 HiPE: Implementation issues

Since we were starting from scratch, we acquired the then-latest Erlang system from Ericsson, version 4.5.3. (This was still before Erlang became open source.) That Erlang system could be configured either as a JAM- or a BEAM-based system at installation time. The BEAM implementation had matured at this point, with a better compiler than JAM², but it was a moving target and had still not proven itself much faster than JAM. Furthermore, we had shown that our straightforward compilation to native code was faster than BEAM. Since basing our compiler on JAM byte-codes was easier (fewer instructions to handle), we chose to stay with the JAM.

Runtime System In order to minimize the number of bugs in this first version of the HiPE system we decided to put some effort into making the interface between emulated and native code as simple and clean as possible. Our performance measurements had shown that reducing the mandatory words of stack frames from four to one (cf., Fig. 1 and 2) gave a performance improvement

² Although technically compilers, they still generate virtual machine code for each system’s emulator.

at least as big as the impact of dataflow optimizations in the JERICO compiler [19] so maintaining the different types of frames was justified, but on the other hand dealing with them in the same memory area was error-prone. In order to avoid these problems, we decided to separate the native code stack from the emulated code stack.³ This way we could have different stack frame formats for emulated and native code, but on either stack we would only have to deal with one type of stack frames. With separate stacks, special effort had to be put into maintaining tail-call optimization as required by Erlang; see Section 4.4. In JERICO, the garbage collector had to keep a state variable indicating the mode (emulated or native) of the stack frame being scanned, and for each frame it had to check if the frame marked a switch to the other mode. With separate stacks, the garbage collector can scan each stack quickly and easily, knowing it will find only one type of frame on each stack. One problem with this scheme is that the exception handling mechanism was implemented by a set of catch frames linked together on the stack; this meant that there are links (pointers) between the two stacks. This causes a slight complication when a stack needs to be relocated (in order to expand or shrink it), since all catch frame links on the other stack must be updated. (We will discuss the integration of emulated and native code further when we describe the current system in Section 4.)

Compiler Rewriting the HiPE compiler in Erlang was not the only step in making it extendible. Instead of the single intermediate format that the JERICO compiler used, several intermediate representation levels were introduced; these are described in detail in Section 4.1. In addition to the optimizations mentioned in Section 3.1.1, the HiPE compiler optimized constant data structures into data references and implemented spilling. Using Erlang and several intermediate formats allowed us to experiment relatively easily and incorporate many compiler optimizations.

On the other hand, while implementing optimization algorithms in Erlang, we experienced performance problems in the HiPE compiler itself. The compiler frequently updated its internal data structures; however, since it was written in Erlang, these “updates” were implemented by creating new versions of the data structures. This spurred us to implement fast declarative arrays and hash tables. These data structures behaved as ordinary immutable Erlang values, but used destructive updates internally, which gave considerable performance benefits.

3.2.2 Instrumentation of HiPE

HiPE’s runtime system is enhanced with performance instrumentation features that can be selectively included

³ Erlang code does not run on the runtime system’s C stack, except when calling primitive BIFs.

or excluded at the system’s installation. These instrumentation features come in two forms:

1. *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.
2. *Performance instrumentation counters (PICs)*: These are based on the Sun UltraSPARC’s performance instrumentation facilities [28]. PICs are made accessible to the user through a built-in 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 [16].

3.2.3 Benchmarking and performance

By 1998, HiPE was slowly becoming a stable system and it was time to measure its performance and the effect of various compiler optimizations on large industrial Erlang applications. We initially attempted to obtain the code for Mobility Server, an Ericsson product for tracking users moving through an enterprise and routing calls to the appropriate point. This did not enjoy much success probably due to lack of direct contact between the HiPE and the Mobility Server development teams.

Luckily, two other Ericsson projects were using Erlang: AXD/301 [4], a scalable ATM switch, and ANx [22], an Asymmetric Digital Subscriber Line (ADSL) system. Both were medium-to-large projects with 100–200 people involved, and their software base consisted of large amounts of Erlang code (i.e., several hundreds of thousands of lines). Furthermore, the developers of AXD/301 were benchmarking conscious—mainly due to the competitiveness of the ATM switch market—and willing to provide “real” data to use as benchmarks for HiPE’s performance evaluation and spend some time explaining how to run these programs. One of the main problems with benchmarking industrial Erlang applications is that they are often connected to a specific hardware and software platform, which is frequently proprietary—most

probably the situation is similar for most embedded control software. In practice, this means that benchmarking has to be conducted on-site, with all the problems that this entails. The AXD/301 benchmark, SCCT, could however be run on a stand-alone workstation. SCCT was a mere 50,000 lines of code, a fifth of the AXD/301 software at that time, but did contain its time-critical portion. Due to our limited resources, we shelved our plans to work with ANx, and began to analyze the AXD/301 code.

Around three months of debugging and extending HiPE were required to make SCCT run successfully—even making SCCT run was in fact non-trivial, despite it being platform independent; for example, the AXD/301 project used an internal version of Erlang not compatible with ours. Our performance measurements are summarized in [16,17]. Basically, we found that the AXD/301 system runs a huge inner loop, spanning hundreds of procedures, none of which stands out in the profile. Because of this, rewriting SCCT in some lower-level language (e.g., in C) is not an attractive option for improving AXD’s performance. On the other hand, due to the size of the program, we found that there is little reuse in the I-cache, and that the system frequently stalls waiting for instructions. Our second finding was that SCCT spends much time inside builtin operations manipulating byte arrays and the internal database. Finally, considerable time is spent in the OS kernel. For these reasons, the performance speedup from compiling SCCT to native rather than to emulated code, although noticeable, is considerably lower than that obtained for small benchmark programs; see also Table 2 in Section 6.2. Still this speedup probably justifies the increased code size and compilation time.

3.3 Open source HiPE

In December 1998, Ericsson released their current Erlang/OTP system as Open Source Erlang (OSE), which opened the possibility of also distributing the HiPE system as Open Source.

Unfortunately, contacts between the two Erlang development groups (HiPE’s and Ericsson’s) had been infrequent and often indirect, which meant that HiPE and Ericsson’s Erlang system had evolved independently for about two years (since Erlang 4.5.3). In short, HiPE was based on an old and obsolete system and had to be ported to OSE before we could release it.

The task of porting HiPE to OSE turned out to be significantly harder than we anticipated (or hoped!). For example, Ericsson’s Erlang system had switched to a different tagging scheme, using the low bits of the word rather than the high ones. The syntactic changes to the Erlang source code since 4.5.3 were massive; our only option was to settle for a mostly manual, and thus extremely slow and painful, “diff & merge” process.

When porting HiPE to the second OSE release, JAM 47.4.1, we were confronted with more surprises! The Ericsson system now featured a *generational* garbage collector [18] which, besides needing modifications for the native code stack, was incompatible with our compiler's use of imperative data structures (see Section 3.2.1). Generational collectors place objects in separate memory areas depending on their age, and they concentrate their efforts to reclaiming memory among the younger objects since they tend to be short-lived. Updates can cause old objects to contain references to young objects. These references need special treatment in most generational collectors: additional data structures are needed to record them, and code must be generated to maintain the data structures [18, Chapter 7.5]. Our problem was that this support did not exist in the runtime system, since the base Erlang system did not need it. At the time we did not have time to implement this support ourselves, so we reverted to using purely functional implementations of the compiler's data structures, which slowed down the compilation times considerably.

The HiPE compiler optimizes constant data structures into references to statically-allocated literals. This too was incompatible with the new generational garbage collector, and we had to change it to explicitly *not* move objects residing in the constant data area. The problem is that an Erlang process' youngest generation is scattered over several distinct memory areas, while its older generation is a single memory area. (It is usually the other way around.) This means that the collector cannot easily test if a pointer refers to the young generation. Instead, it tests if the pointer refers to the older generation, and if not, it assumes that it must point into the young generation.

The BIF calling conventions had changed subtly, and extensions to HiPE's low-level SPARC assembly support code were needed to deal with this.

We also had to track down occasional (and mostly irreproducible) memory corruption bugs in 47.4.1. The tag scheme limits the address range of Erlang values to $[0, 2^{28} - 1]$; consequently, the runtime system has to check that memory blocks allocated for Erlang values actually reside in this "safe" address range. Unfortunately, one allocation site in the runtime system failed to perform this check. When memory usage was high, an unsafe (above 2^{28}) address could be returned. Converting this address to a tagged pointer would lose significant bits, causing later accesses to malfunction as the pointer untagging operation would produce a different address.

On the positive side, the porting effort gave us the opportunity to review and revise some design decisions that in retrospect were not entirely satisfactory. In particular, we re-implemented the mode-switch interface to use new JAM instructions instead of explicit tests (cf., Section 4.2), and tidied up the mode-switch stack frame management (cf., Section 4.4). In addition, we rewrote HiPE's code server and dynamic linker in C. In March

2000, HiPE version 0.92 was finally released as Open Source based on OSE version 47.4.1.⁴ The released system consisted of about 30,000 lines of Erlang code and 3,000 lines of C and assembly code, added to an otherwise mostly unchanged JAM system. Its architecture is described in detail in Section 4.

3.4 Current and future work: HiPE in OSE

HiPE 0.92 has significantly better performance than Ericsson's Open Source Erlang 47.4.1 upon which it is based; Section 6.2 presents detailed performance data that support this claim. One drawback, however, of HiPE 0.92 is that it is based on the JAM emulator. Meanwhile, the BEAM-based system has been improved, and is now reliable and delivers superior performance compared to the JAM. Consequently, Ericsson discontinued the JAM in November 1999. Once again, HiPE became a separate branch in Erlang's development.

Acknowledging this problem, the development teams of HiPE and Erlang/OTP (at Uppsala University and Ericsson, respectively) have since begun tighter cooperation through common system design meetings and frequent exchange of code snapshots. As a first concrete result of this cooperation, some technology transfer from academia to industry has taken place: a data representation that removes the address space limitation in OSE has been incorporated in Ericsson's standard Erlang/OTP system [23]. However, the more ambitious common goal is to include HiPE as a standard component in future releases of Open Source Erlang. To this end, the HiPE compiler's front-end (see Section 4.1) has been replaced with one starting from BEAM bytecodes (another front-end based on Core Erlang [7]⁵ is underway). The Erlang/OTP compiler and object-file loader have also been modified to allow users to easily generate and load native code via the standard compiler and loader interfaces. At the time of making the final touches to this article, May 2001, the integration of the two systems is almost complete and the release of the common system is scheduled for October 2001. We are currently concentrating our efforts on a HiPE back-end for the x86 architecture.

4 HiPE Architecture

The HiPE system consists of a compiler from virtual machine (either JAM or BEAM) code to UltraSPARC machine code, and a runtime system which has been augmented to also support native machine code. This section describes the design and implementation of these components. The architecture of the HiPE system is shown in Fig. 3.

⁴ HiPE can be obtained at www.csd.uu.se/projects/hipe.

⁵ See also <http://www.csd.uu.se/projects/hipe/corer1/>.

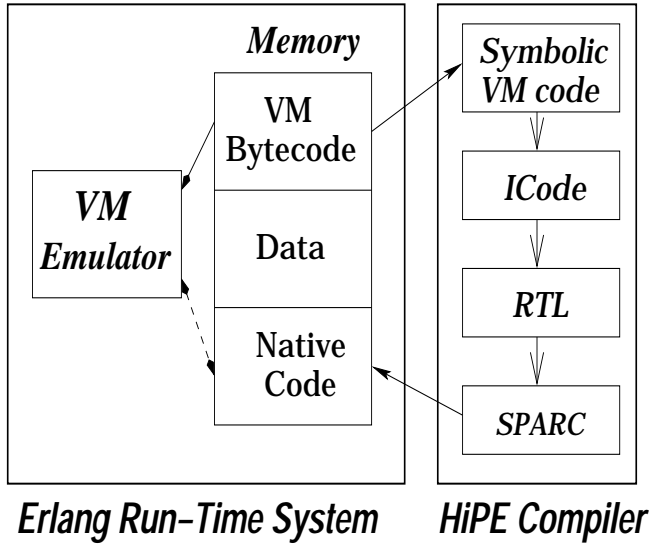


Fig. 3. Intermediate representations in HiPE.

4.1 The HiPE compiler

The HiPE compiler is implemented as a set of Erlang modules. To compile a function, the HiPE compiler is called from Erlang with the function’s descriptor.⁶ The compiler retrieves the function’s bytecodes either from a bytecode file or from memory, translates and optimizes the code via a series of intermediate representations, and finally produces machine code. The machine code is passed on to the HiPE linker, which stores the machine code in memory and integrates it into the running system. Future calls to the compiled function will now invoke the machine-code version. Entire modules can also be compiled, and generated machine code may be saved in files for faster loading in future sessions. The HiPE compiler is further described in [17,19].

4.1.1 Intermediate representations

All of the compiler’s intermediate representations (IRs) are based on control flow graphs (CFGs), where nodes are basic blocks and arcs represent transfers of control. In contrast to ordinary CFGs, the HiPE CFGs can have multiple entry points. This is used to handle exceptions: a procedure has one failure entry point per catch in addition to the normal call entry point. If a callee throws an exception caught by the current function, control will enter the function by one of the failure entry points. The intermediate representations used by the HiPE compiler are:

Icode The Icode IR assumes an infinite number of registers and an implicit stack. There are few primitive op-

⁶ In Erlang, a function is described by a triple $\{module\ name, function\ name, function\ arity\}$. Erlang allows different functions to have the same module and function names, as long as their arities differ.

erators: operations such as arithmetic or constructing data are performed by function calls. Function calls may take any number of parameters, and registers are preserved around calls. Bookkeeping operations, such as heap overflow tests, time-slice tests, and context switching are implicit.

The simplicity and small size of Icode means it is suitable for initial simplifications, type analysis and type optimization, and inline expansion.

RTL The register transfer language RTL is a machine-independent IR similar to three-address code [1] intended to capture conventional compiler optimizations. RTL instructions are similar to the MIPS instruction set.

There are tagged and untagged registers. A tagged register holds a proper Erlang value, while an untagged register may hold an address, a raw integer, or some other value. Untagged registers may not be live across function calls (including calls to the garbage collector), since that would add complexity and inefficiency to the GC. The compiler enforces these rules.

SPARC The final IR is an abstract SPARC assembly language. It adds some symbolic operations (e.g., loading an atom or an address) which are resolved by the linker.

4.1.2 Translation steps

Virtual Machine Code to Icode Each Erlang function is disassembled from bytecode into symbolic virtual machine code and then translated to Icode. In the case of the JAM, which is a pure stack virtual machine while Icode is register-oriented, the translation uses a virtual stack to assign registers to stack slots. Common operations, such as fetching an element from a tuple or pattern-matching, are inline-expanded into fetches and tests. Some obviously poor sequences of virtual machine code are peephole-optimized into more efficient Icode sequences than the concatenation of their individual translations. Message receive operations are translated into loops in Icode.

Icode to RTL In the translation from Icode to RTL, a large number of operations (e.g., arithmetic, data construction, tests) are inline-expanded. Data structure operations are turned into loads and stores. Data tagging and untagging operations [11] are made explicit, to enable optimizations such as constant propagation and folding to be applied to them. Exception handlers are expanded into real code.

After optimization, the RTL IR is rewritten to include stack handling explicitly in the code. At a function call, live variables (tagged registers) are pushed onto the stack; after the call, only variables used before the next call are popped from the stack.

RTL to SPARC Translating RTL to the SPARC IR is straightforward. Some expansion does occur. For instance, the RTL compare-and-jump instruction is im-

plemented by two SPARC instructions (a compare followed by a conditional branch).

A standard graph-coloring register allocator [5] is applied to map RTL registers onto the available SPARC registers. If spilling occurs, a static data area is allocated to the code block for holding spilled temporaries. Register allocation is applied separately to each extended basic block (EBB, a maximal tree in the CFG ending with a control-flow join, return, or function call). Since the runtime system simulates concurrency by scheduling processes at (some) EBB boundaries, and since a function call site ends the EBB, no EBB can have multiple simultaneous activations. Therefore, spilling to a static data area is safe.

The final code is then either stored in a file, or assembled to binary machine code and linked into the runtime system.

4.1.3 Optimizations

HiPE performs a number of common compiler optimizations. The following are applied to the Icode and RTL intermediate representations:

- Unreachable code elimination. Basic blocks which cannot be reached are removed.
- Constant propagation and folding. Constants are propagated, and operators with constant operands are evaluated where it is safe to do so.
- Copy propagation. Eliminate copies $x := y$ by substituting y for x when possible. If x becomes unused as a result, the copy operation is deleted.

Most of the optimizations work on extended basic blocks rather than by fixpoint iteration, in order to save compilation time (an EBB can be analysed and optimized in a single pass).

At the Icode to RTL translation step, heap overflow tests are expanded. Initially, each data allocation site performs its own heap overflow test. To reduce the number of tests needed at runtime, the tests are propagated backwards in the CFG, and adjacent tests are merged (e.g., instead of two separate tests for i and j bytes respectively, test once for $i + j$ bytes).

After translation to SPARC and register allocation, the compiler performs instruction scheduling in order to move useful work into the delay slots of branch instructions.

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 underlying Erlang runtime 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, of the form `module:function(parameters...)`, the JAM emulator first performs a lookup based on the module and function name (in the BEAM emulator, this lookup is optimized). If the function is found, the emulator starts executing its bytecodes; otherwise, an error handler is invoked.

In native code, each function call is implemented as a 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 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.

Figure 4 illustrates the actions of the HiPE linker. Initially, the function `f` exists only as emulated code, and the native code function `g` calls it via a *trap-to-emulated* stub; see Fig. 4(a). After compiling `f` to native code, the call site in `g` is backpatched to invoke `f`'s native code, and the first instruction in `f`'s original emulated code is replaced with a *trap-to-native* emulator instruction; see Fig. 4(b). Fortunately, even the smallest possible JAM or BEAM bytecode function is larger than the *trap-to-native* emulator instruction.

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 bytecodes for that module from the file system. If this is successful, the call continues as normal. As a side-

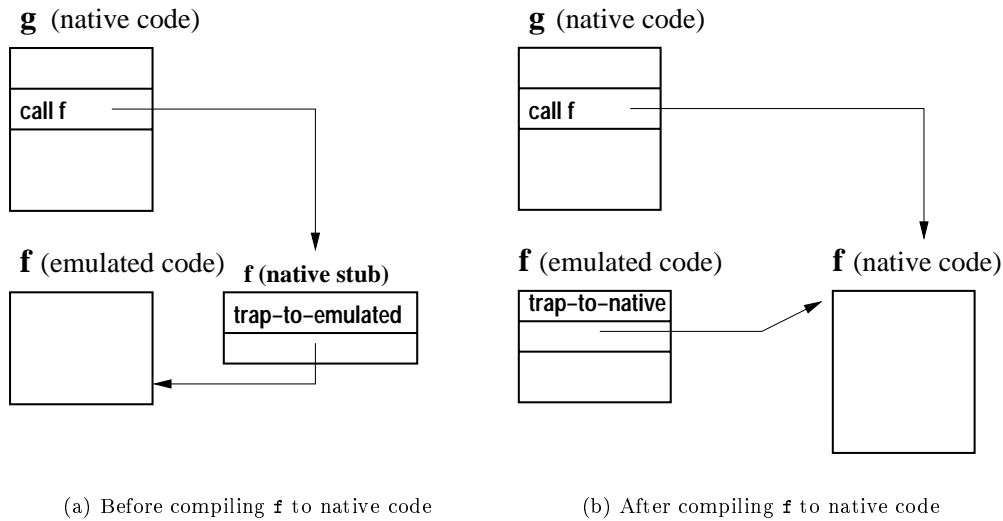


Fig. 4. Code backpatching done by HiPE linker.

effect of loading the 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 code and native SPARC code. Obviously, native and emulated code use the same data representation. However, the emulator’s calling conventions are inappropriate for native code. For instance, JAM passes all parameters on the stack and uses large call frames containing redundant information, as discussed in Section 3.1.1. Instead, native code passes the return address and the first five parameters in registers, remaining parameters (if any) on the stack, and shrinks the fixed portion of stack frames to a single word for preserving the previous return address, as shown in Fig. 2.

HiPE uses two stacks for each process, one for emulated code (the *estack*) and one for native code (the *nstack*). As described in Section 3.2, this is done mainly to simplify the garbage collector, since each stack will only contain frames of a single type. For instance, JAM frames must be scanned in a particular order since some fields are untagged.

Currently, HiPE does not use the SPARC’s register windows; registers are instead saved and restored as needed around function calls. One reason for this is that the windows have fixed size: each adds exactly 16 new registers (24 are visible but 8 are shared with an adjacent window). Many functions do not need all these registers, which means that a number of registers in the register file would contain unpredictable bit-patterns. This, in turn, would make exact garbage collection impossible since our compiler does not yet emit stack-frame layout tables.

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?

The first question which must be answered is: where do mode-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 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?

The second question which must be answered is: how does the system discover that a particular instance of a call, return, or throw event must perform a mode-switch?

Call events. 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.

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 make to the existing emulators.

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 causes 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.

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 emulator. Therefore, when a process is created or resumed, the scheduler simply passes the process’ control block (PCB) to the emulator for execution.

When a process which executes in native code is suspended in HiPE, we set the resume address in the PCB to point to a special emulator instruction. When the scheduler resumes the process, the emulator executes this instruction, which in turn resumes the suspended native code.

Figure 5 illustrates the use of mode-switch frames and return addresses. There are three functions: *f* and *h* are in emulated JAM code, *g* is in native code. First *f* calls *g*, via the trap-to-native instruction planted in *g*’s original emulated code by the linker. At the call, *f*

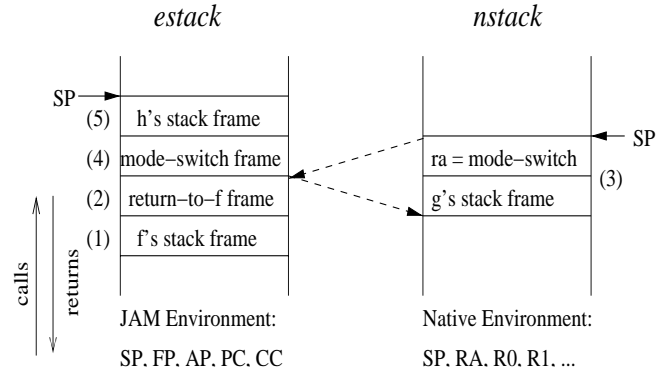


Fig. 5. Mode-switch frames created in call $f \rightarrow g \rightarrow h$.

pushes an emulated-mode return frame (2) on top of its own frame (1). The mode-switch transfers control to *g*, and sets the native-code return address register to point to the native-to-emulated mode-switch routine. Then *g* calls *h*, via *h*’s trap-to-emulated native-code stub. At the call, *g* saves its live registers, including its return address, in frame (3). The mode-switch pushes a mode-switch return frame (4) on the *estack* and invokes *h*, which then creates a frame for its local variables (5). Not shown in the figure are the mode-switch catch frames also created.

4.4.3 Maintaining tail-recursion

The same-mode calling convention with mode-switching 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:

$$f_1^e \xrightarrow{\text{tail}} f_2^n \xrightarrow{\text{tail}} f_3^e \xrightarrow{\text{tail}} f_4^n \xrightarrow{\text{tail}} \dots$$

A correct implementation is expected to execute such a sequence in constant 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 optimization 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:

$$f^e \rightarrow g^n \xrightarrow{\text{tail}} h^e$$

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 tailcalls 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) pop the mode-switch return frame from the caller's stack;
 - (b) pop 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.

The initial test prevents adjacent mode-switches from being created, and thus restores proper tail-recursive behaviour. The test itself is not expensive, and it is only executed when there is a mode-switch call. Similar methods for maintaining proper tail-recursion in the context of a mixed mode execution have been used in some Prolog implementations (e.g., ProLog_by_BIM, SICStus Prolog [24]) and perhaps elsewhere.

4.5 Modifications to the emulators

As described in Sections 4.2–4.4, we have modified the JAM and BEAM emulators to support mixing native and emulated code. In short, 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 process control block, together with a few native-code variables (stack pointer, return 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 emulators, to support mode-switching between emulated and native code.

5 A Critical Examination of some Design Decisions

5.1 The HiPE compiler's intermediate representations

The split into three intermediate representations (IRs) conceptually provides nice abstraction levels: Icode is a simple translation target, RTL can in principle express machine-independent optimizations quite well and is easily retargetable, and the SPARC format is simple and to the point. This separation of concerns has simplified the compiler's development and experimentation with

optimizations. On the other hand, in the current implementation, a number of optimizations such as constant propagation and folding, and dead code elimination are repeated in all IRs; this slows down the compilation process [19].

As described in Section 4.1.1, the compiler's IRs are CFGs with multiple entry points. For a given function, its CFG will have one normal call entry point, and one entry point for each exception handler in its body. In retrospect, this decision makes a number of compiler algorithms more complex. For example, dominators are no longer straightforward to compute, which is an obstacle for SSA-based compiler optimizations [8].

The compiler's RTL IR was intended to be machine-independent, but it is not. Data representation and tagging operations are made explicit in the translation from Icode to RTL. Although a given version of the base Erlang system tends to use the same data representation for all machines, this only applies to proper Erlang values. Native code also needs to manipulate native code return addresses on the native code stack. Since the garbage collector must be able to scan the native stack, native return addresses must be tagged. However, these special tag operations are machine-dependent, e.g., SPARC return addresses are word-aligned while x86 return addresses are not.

An alternative would be to allow the garbage collector to identify which words on the stack are tagged and which words are untagged. This requires support from the compiler and the linker. In particular, it requires generation of meta-data that describe the stack contents. Unfortunately, the current system is not able to do this. This is not a problem for RISC processors, but it needs to be changed before HiPE can support native x86 code.

5.2 Caller-save register management

Register and stack management is performed at two independent places in the HiPE compiler. In the RTL IR, all live registers are saved to the process' native stack around function calls, and in the register allocator, registers are spilled to a static data area. Both the process' native stack and the register spill area are ordinary `malloc()` blocks pointed to by general-purpose registers – neither is related in any way to the processor's stack or stack register.

This arrangement is adequate for RISC machines, but is clearly suboptimal for machines like the x86 which only provide a handful of general-purpose registers. By moving all register management into the register allocator, we could more easily make use of CPU-specific solutions.

For instance, using the processor's stack and stack register for native code on the x86 would permit more efficient code by making one more general-purpose register available to the register allocator or the runtime system. It would also allow the use of the processor's own call

and return instructions. On RISCs, these instructions tend to only use registers, which is safe in HiPE. On the x86 and similar machines, they instead operate on the processor's stack. Modern x86 implementations use the processor's call and return instructions as hints to the branch-prediction hardware. If application code uses ordinary jumps to implement function calls and returns, branch-prediction accuracy will be reduced.

We could also more easily generate register and stack descriptors which describe to the garbage collector which registers are live, where registers have been saved, and which ones contain references to Erlang values. This, in turn, would permit the use of untagged return addresses and callee-save registers.

5.3 Spilling to a static area

We decided to use a static area for spilling since the translation from RTL did not preserve sufficient stack information to easily spill to the stack. Rather than fixing this problem, we circumvented it using a “simple hack” that has somehow remained, warts and all. It is likely that subsequent releases of HiPE will remove this hack.

Until then, spilling to a static area depends on the following system properties:

1. When RTL has been rewritten into SPARC, no register containing an Erlang value survives a function call. In particular, spilled values are not reused across function calls. This means the spill area is empty at a call, and can be reused by recursive activations of the same function.
2. The static area is not scanned by the garbage collector, which is invoked explicitly by the compiled code. Since the spill area is empty at the time of a call, there is nothing to scan and the garbage collector need not know about procedure-specific spill areas.
3. Finally, process scheduling is triggered by a special emulator instruction. This instruction is implemented using a test and a procedure call, which is handled like any other call (viz., by emptying the spill area). Thus, there is no danger due to concurrent activations of the same procedure.⁷

5.4 Compiler optimizations

As mentioned in Section 4.1.3, the main compiler optimizations are not implemented by fixpoint iterations, which is traditional, but by single passes over extended basic blocks (EBBs). This was done in an attempt to speed up the compiler. This was probably a too conservative decision in retrospect: although loops are uncommon in Erlang code, pattern matching generates joins in

⁷ A preemptive multithreaded system would break this property and hence could not use a static spill area.

the CFGs, which cut off EBBs. This limits the effectiveness of these optimizations, especially in code with complex pattern matching, which is common in larger applications. Our experience in adding a *partial redundancy eliminator* [21] to the HiPE compiler confirms this. Due to the above reason, the effectiveness of this optimization was significantly lower than we had come to expect from reports on other languages' optimizing compilers.

5.5 Separate native stack

As described in Section 3.1, an earlier version of HiPE used only one stack for both emulated and native code, but that scheme was eventually abandoned as it was found to be quite complex and difficult to implement correctly. The dual-stack approach has disadvantages and advantages too:

- For example, 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 either stack (to increase its size), then the other stack must also 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.

5.6 New emulator instructions

As described in Section 4.5, HiPE adds a small number of new instructions to the JAM and BEAM emulator, to support mode-switching between emulated and native code.

Prior to the port to Open-Source Erlang 47.4.1, HiPE used dynamic tests in the 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.

5.7 Mode-switching

As described in Section 4.4, HiPE uses a static same-mode approach to mode-switching: each call, return, and

exception throw passes control to code executing on the same mode. The linker and the runtime system insert “software trap” code sequences to trigger mode-switches. HiPE uses this approach due to its performance and simplicity—the initial inspiration is alleged to have come from the implementation of SICStus Prolog.

Other alternatives include dynamic tests and fixed-mode conventions [10]. With dynamic tests, a test is performed at each function call site to determine the mode of the callee. The appropriate code sequence is then selected to perform the call. To implement returns either dynamic testing or mode-switch return frames can be used. With dynamic testing, the return address is tagged to communicate the caller’s type to the callee; at return, the callee inspects the tag (often just a single bit) and chooses the correct return code. With mode-switch return frames, the callee returns in its own mode without any tests: the caller is responsible for pushing an additional mode-switch frame if a mode-switch is necessary (just as in HiPE).

Another common choice is to use a fixed mode for calls, usually native code. Emulated functions are represented as small native code wrappers which invoke the emulator when called. The advantage of this approach is that no dynamic type test is ever needed at a call site. 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, which includes maintaining both emulated and native-code return addresses.

6 A Taste of HiPE’s Performance

The performance characteristics of HiPE have been analyzed in detail before and reported in [16,17]. Here, we first put the performance of Erlang implementations in perspective by comparing it against the performance of other functional languages (Section 6.1), and then in Section 6.2 we briefly report on the performance of the current HiPE system against other implementations of Erlang.

6.1 Erlang vs. other functional languages

Functional programming languages differ significantly in design philosophy (lazy vs. strict, statically vs. dynamically typed), in features they provide (e.g., being concurrent or not), as well as in performance characteristics. For these reasons, comparisons between them cannot be very conclusive. The intention here is to just get a feeling about the performance of Erlang implementations by comparing HiPE (version 0.92) and the JAM system upon which this HiPE version is based (version 47.4.1) against high-performance implementations of other functional languages. Systems used in this comparison are:

Table 1. Performance of Erlang vs. other functional languages.

	qsort	fib	huff	ring(5)
JAM	33.2	144.0	119.2	61.2
HiPE	2.6	16.5	14.8	47.9
Bigloo	6.4	11.7	13.0	—
CML	1.4	17.8	4.4	36.4
CLEAN	0.8	8.8	1.0	—

The Bigloo version 2.1c Scheme compiler [26] (compiling to native code via `gcc -O3`; the Bigloo optimization option `-fstack` was also used), SML/NJ release 110 with the CML extensions [25], and CLEAN version 1.3.2 [6]. Like Erlang, Scheme is a strict, dynamically typed language. CML is concurrent, statically typed, and strict. CLEAN is statically typed and lazy.

This experiment was conducted on a two-processor 248 MHz Sun Ultra-Enterprise 3000 with 1.2 GB of primary memory running Solaris 2.7 using the following four small benchmark programs:

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

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

huff A version of a Huffman encoder. Encodes and decodes a file with 32,026 characters 5 times. The time taken to read the file is not included.

ring This concurrent benchmark creates a ring of 10 processes and sends 100,000 messages. The benchmark is executed 5 times; in Tables 1 and 2, the number of iterations is shown in parentheses. As this benchmark tests the concurrency features of a language, it is run only on implementations that support concurrency.

Performance results (in seconds) are shown in Table 1. As seen, the JAM implementation of Erlang is quite slow compared to implementations of other functional languages; HiPE brings the gap down significantly.

6.2 Comparison of different Erlang implementations

Besides HiPE, four other Erlang systems were used in this comparison: JAM, BEAM, JERICO, and ETOS.

The JAM and BEAM systems used in our measurements are from Ericsson’s Open Source Erlang system upon which HiPE is based. Compared with JAM, the translation of Erlang code to BEAM abstract machine instructions is more advanced. For example, the treatment of pattern matching is considerably better in the BEAM system, even though a full pattern matching compiler is not implemented. Also, BEAM uses a direct-threaded emulator [3] using `gcc`’s labels as first-class objects extension [27]: instructions in the abstract machine code are addresses of the part of the emulator that implement the instruction. The JERICO system has been described in Section 3.1.

ETOS [9] is a system from the University of Montreal based on the Gambit-C Scheme compiler. It translates

Erlang functions to Scheme functions which are then compiled to native code via 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; e.g., simplification of pattern-matching. 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. The version of ETOS used is 2.3.

Most of this performance comparison of Erlang implementations is taken from [17]. It was conducted on a 143 MHz single-processor Sun UltraSPARC 1/140 with 128 MB of primary memory running Solaris 2.6. In addition to **fib**, **qsort**, and **ring**, the following small sequential and concurrent benchmarks were used:

huff_erl A slightly different version of a Huffman encoder compressing and uncompressing a short string 5000 times. The difference from **huff** lies mainly in how the input is provided (for the sake of ETOS which does not currently handle file I/O), but the program is also a bit more Erlang-specific; e.g., it uses polymorphic lists.

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

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

decode Part of a telecommunications protocol. Decodes an incoming binary message 500,000 times. This is a medium-sized benchmark (≈ 400 lines).

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

Besides benchmarks, we also report on the performance of OSE-based systems on two industrial applications of Erlang:

Eddie An HTTP parser handling 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 ATM switch mentioned in Section 3.2.3. It sets up and tears down a number of connections 100 times; 501 functions are used in the benchmark.

Tables 2 and 3 contain the results of the comparison. In all sequential 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. The performance difference between HiPE and ETOS 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.

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` [14] is not very efficient.

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, as mentioned, the **AXD/SCCT** program extensively uses the built-ins to access the shared database on top of the Erlang term storage. As the implementation of these built-ins is currently shared by JAM, BEAM, and HiPE, the percentage of execution spent in these builtins becomes a bottleneck and HiPE's speedup is less than before. Still, HiPE version 0.92 is 24% faster than BEAM on **SCCT**, and considerably faster than the JAM implementation on which it is based.

7 Concluding Remarks

This article described how the HiPE system has been developed, its current architecture, the implementation decisions we had to make, and our experience from our involvement in this several man-year project. Besides documenting our implementation in detail, we believe that the technical issues discussed here are of interest to programming language implementors and hope that our experience proves useful to others interested in Erlang or working in similar projects.

Developing an industrial-strength system in an academic environment is hard in itself. Doing so when parts of the system are evolving independently in industry adds an extra level of complication that ideally should be minimized through interaction. Besides realizing this as a necessity, our experience with working with industrial software development teams in Ericsson has shown that (1) the research group must have direct and frequent contact with the group in industry to build trust and interest, or the transfer of technology and ideas in either direction will be impaired or impossible; (2) despite the difficulties, such collaborations are useful in driving research and in achieving impact in the "real-world".

Acknowledgements. Bjarne Däcker, Håkan Millroth, and Johan Bevemyr were crucial to the initial phase of this project; Bjarne's continued support is appreciated. Christer Jonsson worked on JERICO and the first version of the HiPE compiler; Richard Carlsson is currently working on a Core Erlang front-end. Thomas Lindquist, Ulf Wiger, Kurt Johansson, Mats Cronqvist, and Peter Lundell were extremely helpful with AXD/301. In the Open Source Erlang era, Björn Gustavsson and Kenneth Lundin have been our helpful main contacts. Björn's help in the incorporation of HiPE into OSE

Table 2. Times (in seconds) for benchmarks in different Erlang implementations.

	Sequential Benchmarks						Concurrent		OSE Applications	
	fib	huff_erl	nrev	qsort	smith	decode	ring(100)	life	Eddie	AXD/SCCT
JAM	281.4	234.7	241.3	208.1	114.6	67.8	101.6	13.4	93.6	109.9
BEAM	120.6	69.2	56.9	97.6	53.9	49.0	72.5	8.7	40.0	84.5
JERICO	41.0	14.8	20.5	15.0	25.7	22.5	59.5	7.6	—	—
HiPE	33.8	11.9	18.5	12.3	11.4	22.8	37.1	5.6	18.8	68.0
ETOS	31.8	12.1	24.4	11.0	11.6	52.4	76.0	20.1	—	—

Table 3. Speedup of different Erlang implementations compared to JAM

	Sequential Benchmarks						Concurrent		OSE Applications	
	fib	huff_erl	nrev	qsort	smith	decode	ring(100)	life	Eddie	AXD/SCCT
BEAM	2.33	3.39	4.24	2.13	2.13	1.38	1.40	1.54	2.34	1.30
JERICO	6.86	15.86	11.77	13.87	4.46	3.01	1.71	1.76	—	—
HiPE	8.33	19.72	13.05	16.92	10.05	2.97	2.74	2.39	4.98	1.62
ETOS	8.85	19.40	9.89	18.92	9.88	1.29	1.34	0.67	—	—

is greatly acknowledged. This research has been supported in part by ASTEC (Advanced Software Technology) competence center.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1986.
2. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.
3. J. R. Bell. Threaded code. *Communications of the ACM*, 16(8):370–373, June 1973.
4. S. Blau and J. Rooth. AXD 301—A new generation ATM switching system. *Ericsson Review*, 75(1):10–17, 1998.
5. 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.
6. T. Brus, M. C. J. D. van Eekelen, M. van Leer, M. J. Plasmeijer, and H. P. Barendregt. CLEAN — a language for functional graph rewriting. In Kahn, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA'87)*, number 274 in LNCS, pages 364–384. Springer-Verlag, 1987.
7. R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, and R. Virding. Core Erlang 1.0 language specification. Technical Report 030, Information Technology Department, Uppsala University, Nov. 2000.
8. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. Syst.*, 13(4):451–490, Oct. 1991.
9. 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.
10. A. D. Gordon. How to breed hybrid compilers/interpreters. Technical Report ECS-LFCS-88-50, Department of Computer Science, University of Edinburgh, 1988.
11. D. Gudeman. Representing type information in dynamically typed languages. Technical Report TR 93-27, University of Arizona, Department of Computer Science, Oct. 1993.
12. J. Halén, R. Karlsson, and M. Nilsson. Performance measurements of threads in Java and processes in Erlang. Technical Report ETX/DN/SU-98:024, Ericsson, Nov. 1998.
13. 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.
14. 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.
15. E. Johansson and C. Jonsson. Native code compilation for Erlang. Uppsala master thesis in computer science 100, Uppsala University, Oct. 1996.
16. 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.
17. E. Johansson, M. Pettersson, and K. Sagonas. HiPE: A High Performance Erlang system. In *Proceedings of the ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 32–43. ACM Press, Sept. 2000.
18. R. E. Jones and R. Lins. *Garbage Collection: Algorithms for automatic memory management*. John Wiley & Sons, 1996.
19. T. Lindgren and C. Jonsson. The design and implementation of a high-performance Erlang compiler. Technical Report 99/04, ASTEC, Uppsala University, Nov. 1999.
20. T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification*. The Java Series. Addison-Wesley, 1996.

21. S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufman Publishers, San Fransisco, CA, 1997.
22. P. Nilsson and M. Persson. ANx — high-speed internet access. *Ericsson Review*, 75(1b):24–31, 1998.
23. M. Pettersson. A staged tag scheme for Erlang. Technical Report 029, Information Technology Department, Uppsala University, Nov. 2000.
24. Programming Systems Group. SICStus Prolog User's Manual. Technical report, Swedish Institute of Computer Science, 1995.
25. J. H. Reppy. CML: A higher-order concurrent language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–305. ACM Press, 1991.
26. M. Serrano and P. Weis. Bigloo: a portable and optimizing compiler for strict functional languages. In A. Mycroft, editor, *Proceedings of the 2nd Static Analysis Symposium*, number 983 in LNCS, pages 366–381, Sept. 1995.
27. R. M. Stallman. Using and porting gcc. Technical report, The Free Software Foundation, 1993.
28. Sun Microsystems. UltraSPARC™ User's Manual. Technical report, Sun Microelectronics, Palo Alto, CA, 1997.
29. S. Torstendahl. Open telecom platform. *Ericsson Review*, 75(1):14–17, 1997. See also: <http://www.erlang.se>.
30. 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.
31. D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.