The HiPE/x86 Erlang Compiler: System Description and Performance Evaluation

Mikael Pettersson, Konstantinos Sagonas, and Erik Johansson

Computing Science Department, Uppsala University, Sweden
{mikpe,kostis,happi}@csd.uu.se

Abstract. Erlang is a concurrent functional language, tailored for large-scale distributed and fault-tolerant control software. Its primary implementation is Ericsson's Erlang/OTP system, which is based on a virtual machine interpreter. HiPE (High-Performance Erlang) adds a native code execution mode to the Erlang/OTP system. This paper describes the x86 version of HiPE, including a detailed account of decisions and principles that guide the design of its compiler and runtime system. We also present a brief performance evaluation which indicates that HiPE/x86 delivers performance improvements on par with the more mature HiPE/SPARC system.

1 Introduction

Erlang is a functional programming language which supports concurrency, communication, distribution, fault-tolerance, automatic memory management, and on-line code updates [1]. It was designed for soft real-time control systems which are commonly developed by the telecommunications industry. Judging from commercial applications written in Erlang, the language is quite successful in this domain.

The most widely used implementation of Erlang, Ericsson's Erlang/OTP system, was until recently exclusively based on the BEAM virtual machine interpreter. This, and the fact that Erlang is a dynamically typed language requiring runtime type tests, made Erlang/OTP quite slow compared to implementations of other functional languages. The HiPE system [8, 9] was developed with the aim of significantly reducing this performance gap. HiPE achieves this by allowing flexible, user-controlled, just-in-time compilation of Erlang functions or modules to native machine code. As reported in [8], HiPE is currently the fastest Erlang implementation and offers performance which is competitive with implementations of other strict functional languages such as Scheme or SML/NJ. One drawback was that, until recently, HiPE only supported SPARC-based machines. To enable more widespread use of HiPE, we have developed an x86 version of the system. The result, HiPE/x86, is presented in this paper. Since October 2001, HiPE/x86 is included in the open source release of Erlang/OTP.[1]

In this paper, we present a detailed account of the architecture of the HiPE/x86 compiler and runtime system, design decisions we made and their associated tradeoffs. The purpose of doing so is two-fold: First, we document our implementation in a form which is potentially more easy for other implementors to follow than HiPE's source

[1] Available at www.erlang.org. See also www.csd.uu.se/projects/hipe.

code. Second, we believe that this information is applicable to other garbage-collected tail-recursive high-level languages and as such we hope that our experience will prove useful to anyone that gets involved in a similar project.

The paper begins with a brief overview of Erlang's characteristics (Sect. 2). After presenting the architecture of HiPE (Sect. 3), Sections 4–6 describe the HiPE/x86 system in detail. Some of its planned improvements are also discussed (Sect. 6). Section 7 contains a brief performance evaluation, Section 8 contrasts design decisions with those in related systems, and finally Section 9 ends with some concluding remarks.

2 The Erlang Language and Erlang/OTP

Erlang is a dynamically typed, strict, concurrent functional language. The basic data types include atoms, numbers, and process identifiers; compound data types are lists and tuples. There are no assignments or mutable data structures. Functions are defined as sets of guarded clauses, and clause selection is done by pattern matching. Iterations are expressed as tail-recursive function calls, and Erlang consequently requires tailcall optimisation. Erlang also has a catch/throw-style exception mechanism. Erlang processes are created dynamically, and applications tend to use many of them. Processes communicate through asynchronous message passing: each process has a *mailbox* in which incoming messages are stored, and messages are retrieved from the mailbox by pattern matching. Messages can be arbitrary Erlang values. Erlang implementations must provide automatic memory management, and the soft real-time nature of the language calls for bounded-time garbage collection techniques.

Erlang/OTP is the standard implementation of the language. It combines Erlang with the Open Telecom Platform (OTP) middleware [16], a library with standard components for telecommunications applications. Erlang/OTP is currently used industrially by Ericsson Telecom and other software and telecommunications companies around the world for the development of high-availability servers and networking equipment. Additional information about Erlang can be found at `www.erlang.org`.

3 HiPE: Brief System Overview

HiPE is included as an optional extension in the Open Source Erlang/OTP system. It consists of a compiler from BEAM virtual machine bytecode to native machine code (UltraSPARC or x86), and extensions to the runtime system to support mixing interpreted and native code execution, at the granularity of individual functions.

The overall structure of the HiPE system is shown in Fig. 1. This section briefly describes the generic parts of the compiler and runtime system. The specifics of HiPE/x86 are described in Sect. 4–6. A more detailed system description can be found in [8, 9].

BEAM. The BEAM intermediate representation is a symbolic version of the BEAM virtual machine bytecode, and is produced by disassembling the functions or module being compiled. BEAM operates on a largely implicit heap and call-stack, a set of global registers, and a set of slots in the current stack frame. BEAM is semi-functional: composite values are immutable, but registers and stack slots can be assigned freely.

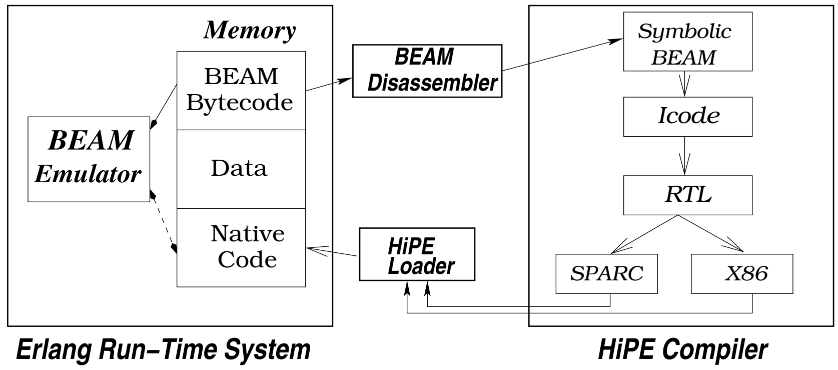

Fig. 1. Structure of a HiPE-enabled Erlang/OTP system.

BEAM to Icode. Icode is an idealised Erlang assembly language. The stack is implicit, any number of temporaries may be used, and temporaries survive function calls. Most computations are expressed as function calls. All bookkeeping operations, including memory management and process scheduling, are implicit.

BEAM is translated to Icode mostly one instruction at a time. However, function calls and the creation of tuples are sequences of instructions in BEAM but single instructions in Icode, requiring the translator to recognise those sequences.

The Icode form is then improved by application of constant propagation, constant folding, and dead-code elimination [12]. Temporaries are also renamed, to avoid false dependencies between different live ranges.

Icode to RTL. RTL is a generic three-address register transfer language. RTL itself is target-independent, but the code is target-specific, due to references to target-specific registers and primitive procedures. RTL has tagged registers for proper Erlang values, and untagged registers for arbitrary machine values. To simplify the garbage collector interface, function calls only preserve live tagged registers.

In the translation from Icode to RTL, many operations (e.g., arithmetic, data construction, or tests) are inlined. Data tagging operations [6, 13] are made explicit, data accesses and initialisations are turned into loads and stores, etc.

Icode-level `switch` instructions for switching on basic values are translated into code that implements the switches [2, 11]. Atoms are problematic since their runtime values differ between invocations of the runtime system, so switches on atoms are translated into semi-symbolic code which is finalised by the object code loader.

Optimisations applied to RTL include common subexpression elimination, constant propagation and folding, and merging of heap overflow tests.

The runtime system. HiPE extends the standard Erlang/OTP runtime system to permit Erlang processes to execute both interpreted code and native machine code. [2]

Each process has two stacks, one for interpreted code and one for native code. This simplifies garbage collection and exception handling, since each stack contains only frames of a single type.

[2] Very few FP language implementations provide support for mixing interpreted and native code.

Control flow between interpreted and native code, e.g. at function calls and returns, is handled by a *mode-switch* interface. The implementation uses linker-generated *proxy code stubs* and *software trap return addresses* to trigger the appropriate mode-switches when invoked. Two important properties of the mode-switch interface are that it preserves tail-recursion (i.e., no sequence of consecutive mode-switching tailcalls grows either stack by more than a constant), and that it imposes no runtime overhead on same-mode calls and returns (i.e., from native to native or from BEAM to BEAM).

4 HiPE/x86 Runtime Architecture

In this section we describe the fundamental aspects of the HiPE/x86 runtime architecture. These are a prerequisite for the compiler's code generator, which is described in Sect. 5. The x86 is a family of CISC-style 32-bit processors. For a compiler writer, these are some of its most relevant properties:

- Arithmetic instructions are in 2-address form, i.e. `dst op= src`.
- It has only 8 general-purpose registers. Although memory operands are permitted in arithmetic instructions, instructions with register operands are faster to decode and execute with shorter latencies.
- Most high-performance implementations use pipelining and out-of-order and speculative execution. Branch-prediction misses are expensive.
- It has built-in support for a call stack, pointed to by the `%esp` general-purpose register, via the `call`, `ret`, `push`, and `pop` instructions.
- The branch-prediction hardware includes a circular return-stack buffer. A `call` instruction also pushes its return address on this buffer, and the predicted target of a `ret` instruction is popped off the buffer.
- For branch instructions with no history recorded in the branch prediction table, there is a static predictor that predicts forwards conditional branches as non-taken, and backwards conditional branches as taken.

Intel [7] in particular considers good branch prediction to be one of the most important factors in achieving high performance.

4.1 Register usage

HiPE/x86 uses `%esp` as the current Erlang process' stack pointer, `%esi` as the current Erlang process' heap pointer, and `%ebp` as the pointer to the process control block (PCB) containing the current process' global state. The remaining 5 registers are used for function parameters and return values, and local caller-save[3] temporaries.

Using `%esp` as the Erlang stack pointer is beneficial since we avoid wasting a register for simulating a stack. It also allows us to use the processor's native support for function calls, although this is a mixed blessing; see Sect. 4.2 and Sect. 4.6.

Global state variables associated with an Erlang process, except for the heap and stack pointers, are accessed as memory operands indirectly via `%ebp`. The state variables frequently accessed this way include the heap and stack limits, and the time-slice counter. While not ideal, most x86 processors handle this fairly well.

[3] Callee-save registers complicate garbage collected implementations, see [4, Sec. 2.3].

4.2 Calling convention

The HiPE/x86 calling convention was designed to meet two main goals: enabling tailcall optimisation (required in Erlang), and avoiding branch prediction misses at returns. To meet the latter goal, we must use the processor's `call` and `ret` instructions, and reserve the `%esp` register for the stack pointer.

Tailcall optimisation is implemented by having the caller *overwrite* its incoming parameter area on the stack with the new parameters for the callee, deallocate the remaining portion of its stack frame, and invoke the callee with a `jmp` instruction. When a function returns, it is also responsible for removing its parameters from the stack; to this end it uses the x86 `ret $`n instruction, which pops the return address, adds n to `%esp`, and then jumps to the return address.

Parameter passing is straightforward: the first 3 parameters are passed in registers, and the remaining ones are pushed on the stack in left-to-right order. A function puts its return value in the `%eax` register.

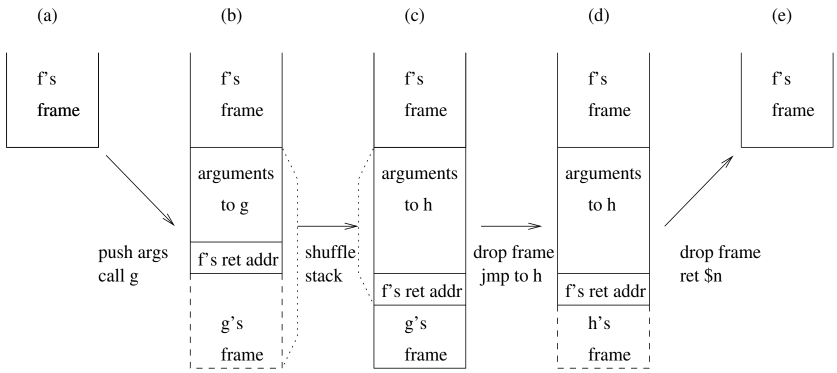

Fig. 2. Recursive calls and tailcalls on x86.

Illustrating calls and tailcalls on x86. To illustrate how recursive calls and tailcalls are implemented by HiPE/x86, assume that `f` recursively calls `g`, `g` tailcalls `h`, and `h` finally returns to `f`. Figure 2 shows the stack layout changes in this process. State (a) shows the stack before `f` calls `g`. `f` evaluates the parameters to `g`, pushes them on the stack, and executes a `call` instruction which pushes a return address and jumps to `g`, leading to state (b). `g` allocates its frame (the dashed portion), evaluates the parameters to `h`, and shuffles the stack to overwrite the argument area and possibly parts of its frame with the new parameters, leading to state (c). Then `g` completes its tailcall to `h` by dropping its frame and jumping to `h`, leading to state (d). In state (d), the stack is exactly as if `f` had called `h` directly. Eventually `h` drops its frame, and executes a `ret $`n instruction, which pops the return address and stacked parameters and returns to `f`, leading to state (e). Register parameters are omitted from this description.

Figure 2 also illustrates why it is the callee which must deallocate the stacked parameters. In the presence of tailcalls, the caller (`f` above) does not know which function finally returns to it, and it does not know how many parameters there currently are on the stack. Therefore, the caller cannot deallocate the stacked parameters, but the returning function *can* since it knows how many parameters *it* takes. We point this out because this is the opposite of the calling convention normally used by C and Unix on x86.

A disadvantage of this calling convention is that the stack shuffle step during tailcalls must also consider the return address as a stacked parameter that will have to be moved to a different stack slot if the caller and callee (`g` and `h` above) have different numbers of stacked parameters. Passing parameters in registers reduces the number of calls with stacked parameters, alleviating this problem.

Tailcalls versus branch prediction. Our calling convention meets its goals of enabling both tailcall optimisation and the processor's return-stack branch predictor, with the downside of sometimes having to copy the return address from one stack slot to another.

An alternative calling convention is to `push` the return address (a code label) before the parameters are pushed, and then `jmp` to the callee. To return, the callee deallocates both its frame and the stacked parameters with a single `add` to the stack pointer, and then performs a simple `ret` or `pops` the return address into a register and performs an indirect `jmp`.[4] The advantage of this calling convention is that the compiler will never have to copy the return address at tailcalls, since it will always be in the slot just above the stacked parameters. However, the return stack branch predictor will not be effective, causing returns to be mispredicted at a higher rate.

As an experiment, we modified the HiPE/x86 assembler to replace `call` instructions with equivalent `push;jmp` sequences. The performance of our benchmark suite dropped by 9.2% overall, with several function call intensive programs losing 20–30%. While we did not utilise this change to improve the tailcall stack shuffle code, it does illustrate the significance of the return stack branch predictor.

Push order. To simplify calling interpreted-mode closures from native code, parameters are pushed left-to-right. In a call to a closure, the closure itself is passed as the *last* parameter (this is the convention for both interpreted and native code), and every interpreted-mode closure has a native-code address which points to a native code stub procedure. Since the parameters are pushed left-to-right, the stub will always find the closure parameter just above the stack pointer, so a single stub can handle all native-code calls to BEAM-mode closures, regardless of how many parameters they take.

Since HiPE/x86 passes 3 parameters in registers, there are actually 4 stubs: three handles cases where the closure ends up as a register parameter, and the fourth handles all cases where the closure is pushed on the stack.

[4] Some language implementations on x86 actually do this; see Sect. 8.

4.3 Stack frames and stack descriptors

A function's stack frame is composed of two parts: a fixed-size part at the top for caller-save registers and spilled temporaries, and a variable-size part at the bottom for pushing the outgoing parameters in recursive calls.

On entry, the function first checks that enough stack space is available for the largest possible frame, calling a runtime system primitive if this is not the case. Then it allocates the fixed-size part by subtracting its size from `%esp`. On return, the function deallocates its frame by adding the frame size to `%esp`, and executing a `ret $`n instruction, which in turn empties the variable-sized part in the caller's frame.

The main benefit of fixed-size frames is their low maintenance cost. On the other hand, they may contain dead or uninitialised stack slots, which can increase their sizes and complicate garbage collection and exception handling.

HiPE/x86 uses *stack descriptors* (also known as *stack maps*) to support exception handling and precise garbage collection. For each call site, the compiler constructs a stack descriptor describing that call site to the runtime system:

- the call site's return address (lookup key)
- the caller's frame size (excluding this call site's actual parameters)
- the caller's arity
- the caller's local exception handler, if present
- the live slots in the caller's frame

This data enables the runtime system to inspect stack frames and traverse call stacks.

A stack descriptor is represented by three-word header containing the return address, hash link, frame size, and arity, followed by a bitmask of the live stack slots. The exception handler is described by a single bit in the header; if set, the handler address is stored in the word immediately *before* the header. The memory overhead for the stack descriptors is currently about 35% of the code size. However, without stack descriptors additional code would have to be generated to remove or nullify dead stack slots.

There are techniques for compacting stack descriptor tables even further, e.g. [15]. However, they also tend to increase stack descriptor lookup costs, and we have found those costs to have a measurable performance impact in our system.

4.4 Garbage collection and generational stack scanning

HiPE/x86 uses a *safe points* garbage collection strategy: the compiler emits code to check for heap overflow and to call the collector in that case. These are normal recursive calls with associated stack descriptors. The garbage collector uses the stack descriptors to traverse the call stack and identify slots containing live Erlang values.

Repeated stack scanning can cause high runtime overheads in deeply recursive programs, so HiPE/x86 implements *generational stack scanning* [4]. Two pointers into the stack are maintained: a *gray* limit and a *black* limit. The black limit indicates how far the stack must be scanned at each minor collection, which corresponds to the boundary between new and old generation data. The gray limit separates frames touched since the previous collection from untouched frames, which corresponds to the boundary between young data and data in the new generation's aging buffer. After a minor collection, data in the aging buffer is promoted to the old generation, the gray limit becomes the black limit, and the gray limit is reset to the bottom of the stack.

To maintain the gray limit we *mark* the frame it points to, which involves saving its return address in the process' control block, and then replacing it with the address of a runtime system primitive. When the code returns to this frame, the runtime primitive moves the mark up a number of frames (currently 4), and then returns to the original return address. (The mark is also updated if an exception is thrown past it.) This implementation differs from [4] in that we never have more than one frame marked gray. Before each garbage collection, we check if the gray limit has passed the black limit, and update the black limit if this is the case. To keep overheads low in common cases, we avoid placing any mark on small stacks, currently stacks less than 512 bytes large. [5]

Generational stack scanning mainly benefits deeply recursive programs. One of our benchmark programs ran 6 times faster with generational stack scanning, and several others ran almost 2 times faster.

4.5 Exception handling

When an exception is thrown, a runtime system primitive is called to find and invoke the handler for the most recently entered protected block. This is done by using the stack descriptors to unwind the stack until a frame with a local handler is found. The stack pointer is reset to the bottom of that frame, and the handler is invoked.

4.6 C and Unix interface issues

Calls between C and Erlang. As described previously, each Erlang process has its own `%esp`-based stack, so when the runtime system (written in C) starts or resumes an Erlang process executing native code, it performs a *stack switch*. Ignoring caching effects, a stack switch is very light-weight, requiring only two instructions to save `%esp` in the PCB and then load its new value.

The Erlang system comes with a large set of built-in functions, BIFs, implemented in C. However, there is no guarantee that the current Erlang process' x86 stack has room for executing a C function. Therefore, to call a BIF, a stack switch back to the standard C stack is made and the actual parameters are copied before the corresponding C function is called – the copy also compensates for differences in parameter passing conventions. After the call, the stacks are switched again and a return is made.

Unix signal handling. A Unix signal handler is typically invoked asynchronously on the current stack. This is problematic for HiPE/x86 since each Erlang process has its own stack. These stacks are initially very small, and grown only in response to explicit stack overflow checks emitted by the compiler. Execution of a signal handler could easily cause stack overflow and memory corruption.

[5] The tuning parameters mentioned here, 4 frames and 512 bytes, were derived from the results of extensive benchmarking.

To avoid this, we redirect all signal handlers to the Unix process' "alternate signal stack". Unfortunately, this redirection is optional and has to be requested as each handler is registered. Since we do not have full control of the entire system in which HiPE/x86 executes (the Erlang/OTP runtime system can be linked with foreign code and libraries), we enforce this by overriding the standard `sigaction()` and `signal()` procedures with our own versions. This is highly system-dependent, which is why HiPE/x86 currently only supports Linux with `glibc` 2.1 or 2.2 libraries, and Solaris 8.

Some user-space threads implementations use over-sized stacks and write-protected guard pages. This is fine for systems supporting < 100 heavy-weight threads in a Unix address space, but an Erlang system is expected to support thousands of light-weight threads, so this solution is not an option for us.

Signals could be masked with `sigprocmask()` before running native code. However, that would increase context switching costs, increase the latency for detecting asynchronous events (such as I/O completion), and make some optimisations implemented in the BEAM interpreter (unmasking floating-point exceptions and catching `SIGFPE` instead of performing dynamic tests) impossible in HiPE/x86.

5 The HiPE/x86 Code Generator

5.1 Backend intermediate representation

The x86 intermediate representation is a simple abstract x86 assembly language. It differs from RTL in two major ways:

- Arithmetic operations are in two-address form, with the destination operand also being a source operand (i.e., `x += y` instead of `x = x + y`).
- Memory operands described by simple addressing modes (base register plus offset) are permitted in most operand positions.

To facilitate conversion to and from control-flow-graph (CFG) form, control-flow instructions with multiple possible successors (e.g., the `jcc` conditional jump instruction) are represented in a pseudo-x86 form with all successors listed explicitly – implicit fallthrough is not used until the final assembly phase.

The translation from RTL to x86 is mainly concerned with converting three-address instructions to two-address instructions, and converting recursive calls to push the actual parameters. The conversion from three-address to two-address form uses the following rules to minimise the the number of new temporaries and moves:

- x = x *op* y becomes x *op*= y
- z = x *op* y becomes z = x ; z *op*= y, if z is neither x nor y
- y = x *op* y becomes y *op*= x, if $y \neq x$ and *op* commutes (e.g., integer $+$)
- y = x *op* y becomes t = x ; t *op*= y ; y = t, if $y \neq x$ and *op* does not commute

Explicit memory operands are introduced in two cases. An RTL-level load instruction becomes a `move` with a memory operand as source; RTL stores are handled analogously. Some global variables located in the PCB are referenced as pseudo-registers at the RTL level, these include the stack limit, heap limit, and the time-slice counter. These pseudo-registers are turned into *offset*`(%ebp)` memory operands.

5.2 Register allocation

Register allocation is applied to try to map temporaries to actual x86 registers. Spilled temporaries remain unallocated, and are mapped to stack slots in the subsequent frame management pass. The default register allocator is based on *linear scan* [14, 10] since it has low compile-time costs and is therefore suitable for JIT compilation, but at higher optimisation levels an *iterated register coalescing* [5] allocator is used.

Before register allocation, every temporary is semantically equivalent to a register, and any operand may be a temporary. After register allocation, any unallocated temporary is an implicit memory operand, but the x86 architecture places some restrictions on the use of memory operands. For example, in binary operations such as arithmetic and moves, either the source or the destination may be a memory operand, but not both. A "fixup" pass is run after register allocation to ensure that these conditions are met.

5.3 Frame management

Stack frames are introduced to the code after register allocation, when the set of spilled temporaries is known. The frame management pass performs the following tasks:

- The register assignment map is applied, and temporaries are rewritten to other temporaries or physical registers.
- A map is built which maps each spilled temporary and stacked parameter to a stack slot. The mapping is given as an offset relative to a virtual frame pointer having the initial value of the stack pointer on entry to the function.
- The frame size and maximal stack usage for calls are computed, and code is added to the function prologue to check for stack overflow and set up the stack frame.
- All instructions are processed while recording the current stack depth (offset from the virtual frame pointer), and references to spilled temporaries are rewritten as memory operands using offsets from `%esp`.
- A stack descriptor is created for each call site, describing which stack slots are live (using the results of a liveness analysis) and whether the call is in the context of a local exception handler.
- At each tailcall, code is generated to copy the actual parameters to the initial portion of the stack frame. Actuals located in the destination area are first pushed at the end of the stack and rewritten as memory operands at those new locations. Then the actuals are moved to the destination area. The return address is treated like any other parameter.

5.4 Linearising the code

During most parts of the compiler, the code is represented in a control-flow graph form. Before assembly, the CFG must be linearised by ordering the basic blocks and redirecting jump instructions accordingly.

It is important that this linearisation takes into account the likelihood of a conditional jump being taken or not, and the static branch prediction algorithm used in current x86 processors (forward conditional: not taken, backward conditional: taken). If this is not done, performance is likely to suffer due to mispredicted branches.

First, the HiPE compiler always annotates conditional jumps with taken/not-taken probabilities. These are usually accurate, since many conditionals are type safety checks and stack or heap overflow checks inserted by the compiler itself. Second, the x86 CFG module is careful to represent all conditional jumps as “unlikely”. Third, the linearisation attempts to order blocks following the “likely” path. The net effect is that in most functions, the likely path is a straight sequence of instructions, with some forward conditional jumps to blocks implementing less likely special cases (such as non-fixnum arithmetic or calling the garbage collector on heap overflow).

5.5 Assembling the code

The assembler converts the final symbolic x86 code to machine code, that is either loaded into the runtime system or saved in an object file. The main complication here is that branch instructions have both short and long forms, with the short forms being preferable for performance reasons but not always applicable. For this reason, assembly is divided into several passes: preliminary un-optimised assembly, rewriting branches to short form when possible, and then final generation of machine code.

6 Planned Future Improvements

Register usage. HiPE/x86 currently passes 3 parameters in registers, but this does not improve performance much because of two limitations: calls clobber all non-reserved registers, and no live range splitting is performed. Since there are no real leaf functions (every function has conditional calls to runtime primitives to handle stack overflow and end of time slice), the parameter registers are always spilled immediately on entry to a function. To rectify this problem, we are planning to perform live-range splitting of temporaries live at function calls, similar to [3].

The frame management pass needs 2 scratch registers while performing the final translation of tailcalls. This is why only 3 parameter registers are used, even though 5 registers are non-reserved in the runtime architecture. To avoid wasting registers in the frame management pass, tailcalls will be reimplemented as parallel moves and expanded before register allocation. The linear scan register allocator also needs 2 scratch registers, but it is not yet known whether it can be fixed to avoid this.

We plan on using the `%fs` segment register instead of a general-purpose register to point to the current process' PCB. This would increase the number of non-reserved registers from 5 to 6. The operating-system support code is working on Linux and Solaris 8, but it is not yet known whether this can be made to work on Windows.

Leaf functions. Every compiled function currently checks for stack overflow when called. By changing the compiler to guarantee that a larger minimum number of words is available on the stack at a call, we should be able to eliminate the stack overflow check from most leaf functions. HiPE/SPARC includes this feature.

Spill slot coalescing. The stack frame module maps each spilled temporary to its own stack slot. Since many temporaries are spilled on x86, it would be advantageous to coalesce spill slots when possible. This would help reduce the size of stack frames, improve cache utilisation, and reduce stack descriptor sizes and the overheads for parsing them.

7 Performance Evaluation

The performance of HiPE on SPARC, both compared with other Erlang systems and with other functional language implementations has been reported previously [8, 9]. Those results showed that HiPE significantly improves the performance of Erlang/OTP on SPARC, and reduces the gap to other functional language implementations. Here we measure the performance of HiPE/x86 vs. BEAM on a set of "standard" Erlang benchmarks, and compare the resulting speedups with those of HiPE/SPARC. The Erlang benchmark programs used are:

fib A recursive Fibonacci function. Calculates `fib(30)` 30 times.
tak Takeuchi function, uses recursion and integer arithmetic intensely. 1,000 repetitions of computing `tak(18,12,6)`.
length A tail-recursive list length function finding the length of a 2,000 element list 50,000 times.
qsort Ordinary quicksort. Sorts a short list 100,000 times.
smith The Smith-Waterman DNA sequence matching algorithm. Matches a sequence against 100 others; all of length 32. This is done 30 times.
huff A Huffman encoder which encodes and decodes a 32,026 character string 5 times.
decode Part of a telecommunications protocol. 500,000 repetitions of decoding an incoming `binary` message. A medium-sized benchmark ($\approx$ 400 lines).
ring A concurrent benchmark which creates a ring of 10 processes and sends 100,000 small messages.
life A concurrent benchmark executing 10,000 generations in Conway's game of life on a 10 by 10 board where each square is implemented as a process.
prettypr Formats a large source program for pretty-printing, repeated 4 times. Recurses very deeply. A medium-sized benchmark ($\approx 1,100$ lines).
estone Computes an Erlang system's Estone number by running a number of common Erlang tasks and reporting a weighted ranking of its performance on these tasks. This benchmark stresses all parts of an Erlang implementation, including its runtime system and concurrency primitives.

Table 1 shows execution times (in milliseconds) for running these benchmarks in Erlang/OTP R8 under BEAM and HiPE, and the speedup for HiPE over BEAM. For the **estone** benchmark, execution times are not shown as the benchmark contains artificial delays. Instead, the speedup is the relative improvement to the Estone-ranking for HiPE over BEAM. The performance evaluation was conducted on a Dell Inspiron 7000, with a 333 MHz Pentium-II processor, 128 MB memory, running Linux. For comparison, the last column shows the speedup of HiPE over BEAM on a Sun Enterprise 3000 with two 248 MHz UltraSPARC-II processors, 1.2 GB memory, running Solaris 7.

Table 1. HiPE/x86 speedup compared with BEAM and HiPE/SPARC.

	x86			SPARC
Benchmark	BEAM	HiPE	speedup	speedup
fib	31230	7360	4.24	4.12
tak	31900	5570	5.73	5.68
length	20000	3540	5.65	9.30
qsort	16450	3630	4.53	4.14
smith	23830	4960	4.80	3.69
huff	24540	12220	2.01	3.14
decode	19410	5210	3.73	2.22
ring	18360	16840	1.09	0.99
life	25580	18690	1.37	1.30
prettypr	13830	4740	2.92	2.37
estone			2.25	2.31

HiPE/x86 achieves speedups comparable to those of HiPE/SPARC, even though the x86 is a register-starved architecture. As explained in Sect. 6, HiPE/x86 currently spills the parameter registers on entry to a function. This hurts the **length** and **huff** benchmarks, since they frequently dereference pointer parameters to list cells on the heap. **fib** and **tak** are also function-call intensive, but suffer less since they operate on integer parameters that can be referenced directly as memory operands on the stack.

HiPE's modest speedup on **ring** and **life** is because they spend most of their time in the runtime system scheduling processes, and compute very little on their own.

8 Related Systems

We have investigated several other native x86 implementations of languages with similar requirements as Erlang, such as ML, Lisp, Scheme, Prolog, Clean, and Haskell. Our investigation focused on how these systems implement calls and returns, and how many registers are used for parameter passing. We found most systems to have inadequate documentation about their implementation (a tradition we intend to break with this paper), so we had to try to derive this information from their source code. This was not always successful, so the findings we report here are somewhat approximate, and we have excluded systems whose source code was too impenetrable.

Table 2 summarises our findings. The "native" column is checked for systems that use `call` and `ret` instructions on a normal `%esp`-based stack, which signifies that return-stack branch prediction is enabled. The "simulated" column is checked for systems that use direct and indirect `jmp` instructions for calls and returns, which signifies that returns risk causing branch prediction misses. The last column indicates how many registers are used for passing function parameters.

The three top-most systems listed (Objective Caml, Clean, CMU Common Lisp) are known to deliver relatively good performance, and it is interesting to note that all three use native call stacks and pass parameters in registers like HiPE/x86. MLKit and MIT Scheme use `%esp` as their stack pointer, but choose to simulate calls and returns, presumably in order to reduce the amount of stack rewriting at tailcalls.

Table 2. Feature summary of related language implementations on the x86.

	call stack		argument
	native	simulated	registers
Objective Caml 3.04	X		6
Clean 2.0	X		4
CMU Common Lisp 20020228	X		3
Poplog V15.53	X		3
Poly/ML 4.1.2	X		2
Chalmers Haskell 0.9999.4	X		0
MLton 20011006		X	0
SML/NJ 110.35		X	1–5
MLKit 4.0		X	3
MIT Scheme 7.6.1		X	0
GNU Prolog 1.2.8		X	0

SML/NJ uses a single argument register in its standard calling convention, since Standard ML functions always have exactly one formal parameter; multiple parameters are expressed as tuples or curried applications. Non-standard calling conventions can be used for local functions, and then up to 5 parameter registers may be available.

In MLton, `%esp` is the heap pointer and `%ebp` points to a simulated call stack. Using a different register than `%esp` to point to the stack is motivated [6] by two main reasons: the MLton stack grows towards higher addresses, which prohibits the use of `push`, `call`, and `ret`, and the compiler generates many accesses to variables in stack frame slots, but the instruction encoding when `%esp` is the base pointer is one byte longer than for other registers, leading to worse code density. Since `%esp` was not used for the stack, it was instead used as the heap pointer. `sigaltstack()` is used to redirect signal handlers, but this does not work on Windows, forcing MLton to ignore `%esp` and use another register as heap pointer on Windows.

Poly/ML also uses `sigaltstack()` to redirect signal handlers. Their workaround on Windows is to artificially enlarge the stack with a scratch area at the bottom.

9 Concluding Remarks

This paper has described the architecture and implementation of HiPE/x86, a publicly available native code compiler for Erlang. HiPE/x86 offers flexible, user-controlled integration of interpreted and native code, and allows significant performance improvements for Erlang applications on the widely available standard x86 platform.

Besides documenting our implementation choices, we believe that such information is potentially useful for anybody that is involved in the implementation of a tail-recursive high-level language that relies on precise garbage collection. As shown by the performance evaluation, the x86 port of HiPE manages to achieve performance improvements compared to interpreted BEAM code which are on par with the more

[6] Matthew Fluet, email correspondence, May 8, 2002.

mature HiPE/SPARC system. We believe that this validates our design choices, especially the function call and stack frame conventions which are surprisingly conventional compared to those in HiPE/SPARC.

Acknowledgements

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

References

1. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.
2. R. L. Bernstein. Producing good code for the case statement. *Software – Practice and Experience*, 15(10):1021–1024, Oct. 1985.
3. R. G. Burger, O. Waddell, and R. K. Dybvig. Register allocation using lazy saves, eager restores, and greedy shuffling. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 130–138. ACM Press, 1995.
4. P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'98*, pages 162–173. ACM Press, 1998.
5. L. George and A. W. Appel. Iterated register coalescing. *ACM Trans. Prog. Lang. Syst.*, 18(3):300–324, May 1996.
6. D. Gudeman. Representing type information in dynamically typed languages. Technical Report TR 93-27, University of Arizona, Department of Computer Science, Oct. 1993.
7. Intel Corporation. *Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual*, 2002. Document number 248966-05.
8. 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.
9. E. Johansson, M. Pettersson, K. Sagonas, and T. Lindgren. The development of the HiPE system: Design and experience report. *Springer International Journal of Software Tools for Technology Transfer*, 2002. To appear.
10. E. Johansson and K. Sagonas. Linear scan register allocation in a high performance Erlang compiler. In *Practical Applications of Declarative Languages: Proceedings of the PADL'2002 Symposium*, number 2257 in LNCS, pages 299–317. Springer, Jan. 2002.
11. S. Kannan and T. A. Proebsting. Correction to "Producing good code for the case statement". *Software – Practice and Experience*, 24(2):233, Feb. 1994.
12. S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufman Publishers, San Fransisco, CA, 1997.
13. M. Pettersson. A staged tag scheme for Erlang. Technical Report 029, Information Technology Department, Uppsala University, Nov. 2000.
14. M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Trans. Prog. Lang. Syst.*, 21(5):895–913, Sept. 1999.
15. D. Tarditi. Compact garbage collection tables. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*, pages 50–58. ACM Press, Oct. 2000.
16. S. Torstendahl. Open Telecom Platform. *Ericsson Review*, 75(1):14–17, 1997.