HiPE on AMD64

Daniel Luna
luna@update.uu.se

Mikael Pettersson
mikpe@it.uu.se

Konstantinos Sagonas
kostis@it.uu.se

Computing Science
Department of Information Technology
Uppsala University, Sweden

ABSTRACT

Erlang is a concurrent functional language designed for developing large-scale, distributed, fault-tolerant systems. The primary implementation of the language is the Erlang/OTP system from Ericsson. Even though Erlang/OTP is by default based on a virtual machine interpreter, it nowadays also includes the HiPE (High Performance Erlang) native code compiler as a fully integrated component.

This paper describes the recently developed port of HiPE to the AMD64 architecture. We discuss technical issues that had to be addressed when developing the port, decisions we took and why, and report on the speedups (compared with BEAM) which HiPE/AMD64 achieves across a range of Erlang programs and how these compare with speedups for the more mature SPARC and x86 back-ends.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Code generation*; D.3.4 [**Programming Languages**]: Processors—*Incremental compilers*; D.3.4 [**Programming Languages**]: Processors—*Run-time environments*; D.3.2 [**Programming Languages**]: Language Classifications—*Applicative (functional) languages*

General Terms

Experimentation, Measurement, Performance

Keywords

Erlang, native code compilation, AMD64

1. INTRODUCTION

Erlang is a functional programming language which efficiently supports concurrency, communication, distribution, fault-tolerance, automatic memory management, and on-line code updates [3]. It was designed to ease the development of soft real-time control systems which are commonly developed by the telecommunications industry. Judging from commercial applications written in Erlang and the increased interest in the language, as witnessed e.g. by the number of downloads and the level of activity on the Erlang mailing list, the language is quite successful in this domain.

The most widely used implementation of the Erlang language, the Erlang/OTP system from Ericsson, has been, till recently, exclusively based on the BEAM virtual machine interpreter. This and the fact that Erlang is a dynamically typed language requiring runtime type tests make the performance of Erlang programs quite slow compared with “equivalent” programs written in other functional languages.

The HiPE native code compiler [9, 10][1] has been developed with the aim of reducing this performance gap. It achieves this goal while allowing fine-grained, user-controlled compilation of Erlang functions or modules to native machine code. As reported in [9], HiPE is currently the fastest Erlang implementation and offers performance which is competitive with other implementations of strict functional languages such as Bigloo Scheme or SML/NJ.

Before we started working on the project whose results we report here, the latest open source release of Erlang/OTP was R9C-0.[2] The HiPE compiler is an integrated component in that release and its available back-ends are for SPARC V8+ and for x86, both running in 32-bit mode. As 64-bit architectures offer significant advantages for large-scale applications (e.g., they allow a much less restricted address space) and are becoming more and more widespread, we wanted to develop a 64-bit back-end for HiPE. Although we briefly considered IA-64 and SPARC64, our chosen platform was the AMD64. Reasons for this choice were the similarity of this platform with the x86 (see Section 2), its upcoming importance,[3] and the affordability of these machines. We have developed an AMD64 back-end for HiPE, called HiPE/AMD64, running on Linux, and we report on aspects of its implementation in this paper.

Contributions. Our first contribution is indirect and only briefly described in the paper: As a by-product of our engagement in this feat, we have cleaned up the Erlang/OTP runtime system and improved its performance on 64-bit architectures. Our second contribution is technical: We describe in detail the architecture, its design decisions, and

Erlang'04, September 22, 2004, Snowbird, Utah, USA.
Copyright 2004 ACM 1-58113-918-7/04/0009 ...$5.00.

[1] See also HiPE's homepage: `www.it.uu.se/research/group/hipe/`.

[2] Available at `www.erlang.org`.

[3] About three months after we literally assembled an AMD Athlon 64 based PC by buying its parts individually on the net, Sun announced its Sun Fire V20z server, the first in a new line of AMD Opteron 64 based servers from Sun; see `www.sun.com/amd/`.

solutions to technical issues that had to be addressed for the development of HiPE/AMD64, and report on its performance. Our final contribution, probably of more interest to the vast majority of the Erlang community, is the system itself which will be included in the upcoming open source release of Erlang/OTP R10.

Organization. The rest of this paper starts with a brief overview of characteristics of the AMD64 architecture (Section 2) and continues with reviewing the organization of the HiPE compiler in Section 3. The main part of this paper describes the HiPE/AMD64 back-end in detail (Section 5) and the support it requires from the runtime system of Erlang/OTP (Section 4). Section 6 contains performance results, while in Section 7 we list some advantages of disadvantages of AMD64 from the perspective of an Erlang user. Finally, Section 8 finishes with some concluding remarks.

2. AN OVERVIEW OF AMD64

AMD64 is a family of general-purpose processors for server, workstation, desktop, and notebook computers [1].

Architecturally, these processors are 64-bit machines, with 64-bit registers (16 integer and 16 floating-point) and 64-bit virtual address spaces. An important feature is that they are fully compatible with 32-bit x86 code. AMD64 processors can run 32-bit operating systems and applications, 64-bit operating systems and applications, or 64-bit operating systems with 32-bit applications.

A distinguishing implementation feature of the current AMD64 processors is their integrated memory controllers, which increase bandwidth and reduce latencies for memory accesses. Another implementation feature is that the server processors support multiprocessing (up to 8-way) without the need for external support components, which reduces the cost and complexity of such systems.

Although the design originated from AMD, Intel has announced that it will release software-compatible 64-bit processors in the near future.[4]

2.1 Technical Summary

Here we summarize the technical aspects of AMD64 that are relevant for compiler writers. Many of these are shared with x86; differences from x86 are described later.

- Instructions are in 2-address form, i.e. `dst op= src`. Although operating on registers is generally faster, most instructions allow either `dst` or `src`, but not both, to be memory operands. A memory operand is the sum of a base register, a scaled index register, and a constant offset, where most parts can be omitted.

- The AMD64 has 16 general-purpose registers and 16 floating-point registers, all 64-bit wide. Instructions on 32-bit integers automatically zero-extend their results to 64 bits (32-bit operands are default on AMD64), while instructions on 16 or 8-bit integers leave the higher bits unchanged.

- The implementations use pipelining, and out-of-order and speculative execution of instructions; this means that branch prediction misses are expensive.

- The dynamic branch prediction hardware has a buffer that remembers whether a given branch is likely to be taken or not. When a branch is not listed in the buffer, the static predictor assumes that backward branches are taken, and forward branches are not taken.

- There is direct support for a call stack, pointed to by the `%rsp` general purpose register, via the `call`, `ret`, `push` and `pop` instructions.

- The return stack branch predictor has a small circular buffer for return addresses. A `call` instruction pushes its return address both on the stack and on this buffer. At a `ret` instruction, the top-most element is popped off the buffer and used as the predicted target of the instruction.

- Instructions vary in size, from one up to fifteen bytes. The actual instructions opcodes are usually one or two bytes long, with prefixes and suffixes making up the rest. Prefixes alter the behaviour of an instruction, while suffixes encode its operands.

For code optimizations, there are a few general but important rules to obey (cf. also [2]):

1. Enable good branch prediction. Arrange code to follow the static branch predictor's rules. Ensure that each `ret` instruction is preceded by a corresponding `call` instruction: do not bypass the call stack or manipulate the return addresses within it.

2. Many instructions have different possible binary encodings. In general, the shortest encoding maximizes performance.

3. Keep variables permanently in registers when possible. If this is not possible, it is generally better to use memory operands in instructions than to read variables into temporary registers before each use.

4. Ensure that memory accesses are to addresses that are a multiple of the size of the access: a 32-bit read or write should be to an address that is a multiple of 4 bytes. Reads and writes to a given memory area should match in address and access size.

2.2 Differences from x86

The main differences from x86, apart from widening registers and virtual addresses from 32 to 64 bits and doubling the number of registers, concern instruction encoding, elimination of some x86 restrictions on byte operations, and the new floating-point model.

The x86 instruction encoding is limited to 3 bits for register numbers, and 32 bits for immediate operands such as code or data addresses. AMD64 follows the x86 encoding, with one main difference: the REX prefix.

The REX prefix, when present, immediately precedes the instruction's first opcode byte. It has four one-bit fields, W, R, X, and B, that augment the instruction's x86 encoding. As mentioned previously, even though AMD64 is a 64-bit architecture, most instructions take 32-bit operands as default. The W bit in the REX prefix changes instructions to use 64-bit operands. The R, X, and B bits provide a fourth (high) bit in register number encodings, allowing access to

[4] Intel calls this Intel® EM64T (Extended Memory 64 Technology) though; see `www.intel.com/technology/64bitextensions/`.

the 8 new registers not available in the x86. The REX prefix uses the opcodes that x86 uses for single-byte `inc` and `dec` instructions. On AMD64, these instructions must use a two-byte encoding.

On AMD64 immediate operands, such as address or data constants, are limited to 32 bits just as on x86. This means that branches, calls, and memory accesses cannot directly access arbitrary locations in the 64-bit address space; such accesses must in general be indirect via a pointer register. To simplify the construction of 64-bit constants, AMD64 has a new instruction which takes a 64-bit immediate operand and copies it into a specific register.

On AMD64 any general purpose register can be used in a load or store operation accessing its low 8 bits. On x86 only registers 0–3 can be used in this way, since register numbers 4–7 actually denote bits 8 to 15 in these registers in byte memory access instructions.[5]

Every AMD64 processor implements the SSE2 floating-point instruction set, which is register-oriented with 16 registers. x86 processors have traditionally used the `x87` instruction set, which is based on an 8-entry stack. SSE2 benefits a compiler mainly because it avoids the restrictions of the `x87` stack; the 8 additional registers also help in code with many floating-point variables.

3. HiPE: A BRIEF OVERVIEW

Since October 2001, HiPE is included as an integrated component in the open source Erlang/OTP system. A high-level view of its current architecture is shown in Figure 1. As far as the Erlang/OTP system is concerned, the HiPE component consists of three main parts:

1. the HiPE compiler which translates BEAM virtual machine bytecode to native machine code in either a just-in-time or an ahead-of-time fashion;
2. the HiPE loader which loads the generated native code on-the-fly or from a fat `.beam` file into the code area; and
3. a set of extensions to the Erlang/OTP runtime system to efficiently support mixing interpreted and native code execution, at the granularity of individual Erlang functions.

In order to make this paper relatively self-contained, this section briefly describes these parts. A more detailed system description of HiPE can be found in [9, 10].

3.1 The HiPE Compiler

Currently, compilation to native code starts by disassembling the bytecode generated by the BEAM compiler, and representing it in the form of a symbolic version of the BEAM virtual machine bytecode. This version is then translated to Icode, which is an idealized 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. Icode is internally represented in the form of a *control flow graph* (CFG). In this stage various optimizations are done. First, there are passes that handle some of the inlining of binary operations [8] and add code for handling exceptions. Then the CFG is turned into SSA form [6], where HiPE performs sparse conditional constant propagation [18], dead code removal, and copy propagation [12]. Finally, a type propagator eliminates type tests whose outcome is statically determined, or pushes these tests *forward* in the CFG to the point that they are really needed.

Icode is then translated into RTL, which is a generic (i.e., target-independent) three-address register transfer language, but the code is target-specific, mainly due to the use of platform-specific registers when accessing a process' state, differences in address computations, and some differences in the built-in primitives. At this level almost all operations are made explicit. For example, data tagging and untagging is translated to appropriate machine operations (**`shift`**, **`or`**, *etc*), data accesses are turned into loads and stores. Also arithmetic operations, data constructions and type tests are inlined.

Finally, RTL code is translated to the target back-end. As shown in Figure 1, currently available back-ends of the HiPE compiler are SPARC V8+, x86, or AMD64. The AMD64 back-end is described in Section 5.

Additions and changes for AMD64 on RTL. Although performed at the level of a register transfer language, the handling of arithmetic is word-size specific. For this reason, a new module of the HiPE compiler was developed which performs 64-bit arithmetic. Although conceptually this is trivial, its implementation turned out quite tricky given that "fixnums" are larger on AMD64.[6]

Various other cleanups were required: Although in principle target-independent, RTL was contaminated by various implicit assumptions about word size, which were unnoticed on 32-bit machines. The offending places of RTL code had to be identified, factored out, and moved to a generic service module, parameterized by the target. For instance, all bitmasks in tagging operations had to be extended to be 64 bits long, and some data structures needed to have their size in words changed (most notably 64-bit floating point numbers). Similarly, field offset computations needed to be parametrized by the word size. More cleanups of this kind were required for the creation of jump tables used in the pattern matching compilation of `case` expressions.

3.2 The HiPE Loader

The HiPE loader is responsible for loading native code into the code area of the Erlang/OTP runtime system and for patching call sites in code which is already loaded to call this code. Special care needs to be taken in order to preserve the semantics of code replacement in Erlang, especially in cases where interpreted and native code is freely intermixed; see [10, Section 4.2] on how this is done. For AMD64, the x86 loader was cloned and the required changes were limited to being able to write 64-bit rather than 32-bit constants to memory.

3.3 Extensions to the Runtime System

HiPE extends the standard Erlang/OTP runtime system to permit Erlang processes to execute both interpreted code and native machine code. In this respect, HiPE is probably

[5] In HiPE/x86, this restriction is currently worked around by always using the `%eax` register in byte memory accesses.

[6] Erlang/OTP fixnums are integer values that fit in one word and thus, with a 4 bit type tag, are 28 bits long on a 32-bit machine and 60 bits long on a 64-bit machine.

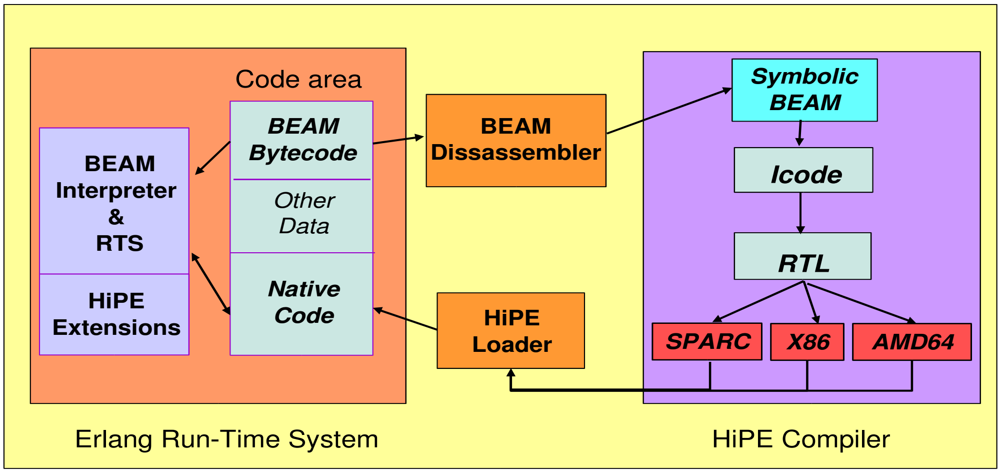

Figure 1: Architecture of a HiPE-enabled Erlang/OTP system.

unique: We know of no other functional programming language implementations that support mixing interpreted and native code in an arbitrary way.

Each Erlang process has two stacks, one for interpreted code and one for native code. As explained in [10], this simplifies garbage collection and exception handling, since each stack contains only frames of a single type. 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 tail calls 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).

Changes to the runtime system for AMD64 are described in the next section.

4. THE AMD64 RUNTIME SYSTEM

The Erlang/OTP runtime system is written in C, and consists of the BEAM virtual machine interpreter, the garbage collector, the Erlang process scheduler, and a large number of standard Erlang functions implemented in C (the BIFs).

Large parts of the runtime system are written in machine-independent C code and did not need changes for AMD64; these include the mode-switch interface, BIFs used by the native-code loader, BIFs used for debugging and performance measurements, and some primitive operations used by compiled code.

Other parts of the runtime system are machine-specific:

- The low-level code for transitions between the mode-switch interface and compiled Erlang machine code. This is implemented in hand-written assembly code, invoked via a small interface written in C.
- The glue code for calling C BIFs from compiled Erlang machine code. This is assembly code, generated by running an `m4` script on the list of BIFs.
- The code to traverse the call stack for garbage collection and for locating exception handlers. This is implemented in C.
- Creating native code stubs for Erlang functions that have not yet been compiled to native code. This is implemented in C.
- Applying certain types of patches to native code during code loading. This is implemented in C.

In the AMD64 port, the C code for traversing the call stack, as well as the C code between the mode-switch interface and the low-level assembly glue code, is shared with the x86 port. This is possible because the only relevant difference between AMD64 and x86 in this code is the word size, and carefully written C code can adapt to that automatically.

The C code for creating stubs and for applying patches was rewritten for AMD64. In both cases this is because the code creates or patches AMD64 instructions containing 64-bit immediates.

The `m4` script generating assembly wrapper code around C BIFs was rewritten for AMD64. The main reason for this is that the wrappers are highly dependent on C's calling convention, and C uses different calling conventions on x86 and AMD64: on x86 all parameters are passed on the stack, while on AMD64 the first six are passed in registers.[7]

The low-level glue code between the mode-switch interface and compiled Erlang machine code was rewritten for AMD64. Since this code is both called from C and calls C, it is dependent on C's calling conventions. There are also some syntactic differences between x86 and AMD64 related to the use of 64-bit operands and additional registers.

A Unix signal handler is typically invoked asynchronously on the current stack. This is problematic for HiPE/AMD64 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. To avoid stack overflow due to signals, we redirect all signal handlers to the Unix process' "alternate signal stack", by overriding the standard `sigaction()` and `signal()` procedures with our own versions. Doing this is highly system-dependent,

[7] For more information see `www.x86-64.org/abi.pdf`

which is why HiPE/AMD64 currently only supports Linux with recent `glibc` libraries. These issues are shared with HiPE/x86, see [14] for more information.

HiPE/AMD64 shares roughly one third of its runtime system code with HiPE/x86. The remaining two thirds were copied from HiPE/x86 and then modified as described above.

4.1 64-bit cleanups in Erlang/OTP

The common Erlang/OTP runtime system was, prior to our work on AMD64, believed to be 64-bit clean. However, we discovered some limitations which we were forced to eliminate. In particular, although Erlang term "handles" and pointers to data had been widened to 64 bits, the representation of integers (both small fixed-size integers and heap-allocated "big" integers) was unchanged from the 32-bit runtime system. This caused a major problem:

Efficient native code arithmetic relies on using the processor's overflow flag to detect when fixnums must be converted to bignums. For this to work, fixnums must be as wide as machine words (minus the type tag), but Erlang still used 28-bit (32 bits minus 4 tag bits) fixnums on 64-bit machines. Failure to detect overflow in the 28-bit representation made native code produce fixnums where bignums should have been produced, causing problems when fixnums were passed from native code to BEAM.

To eliminate this problem we modified the Erlang/OTP runtime system to use word-sized fixnums also on 64-bit machines. Due to undocumented dependencies between the representations of fixnums and bignums, and assumptions in the code for hashing Erlang terms and for converting terms to and from the external binary format, this required a significant amount of work. In addition to being able to support native code on AMD64, the result is a cleaner runtime system with less overhead and higher performance on 64-bit machines. These changes will be part of the next release of Erlang/OTP.

5. THE AMD64 BACK-END

The phases of the AMD64 back-end are shown in Figure 2. In the HiPE compiler, the AMD64 intermediate representation is a simple symbolic 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 are allowed, in the form of base register + register or constant.

Since the intermediate representation is based on control flow graphs instead of linear code, calls and conditional branch instructions are pseudo-instructions that list all their possible destinations. These pseudo-instructions are converted to proper AMD64 instructions just before the code is passed to the assembler; see Section 5.5.

5.1 RTL to AMD64 Translation

The conversion from RTL to AMD64 is mainly concerned with converting RTL's three-address instructions to two-address form. This procedure is the same as for the x86 and is described in [14].

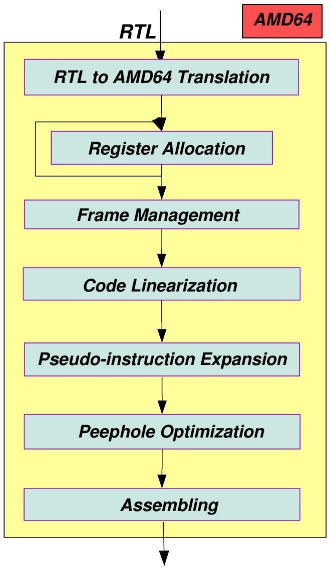

Figure 2: The AMD64 back-end of Fig. 1 in detail.

One problem is that AMD64 instructions cannot have immediate operands larger than 32 bits, with the exception of the new `mov reg imm64` instruction. Therefore, when a 64-bit constant occurs in an RTL instruction, a new temporary register is allocated and code is generated to copy the constant into the register. This must be done also for symbolic constants that denote runtime addresses, such as references to jump tables. This problem does not exist on x86 since its 32-bit immediates perfectly match its 32-bit word size.

5.2 Register Allocation

After translation from RTL, register allocation is performed to map the usually large number of temporaries (pseudo-registers) on to the machine's actual registers.

Register allocation is typically performed in a loop. First an attempt is made to allocate registers for the code. If this fails because some temporaries were spilled (could not be assigned to registers), the code is rewritten under the assumption that those temporaries are in memory, and the process continues with a new allocation attempt. Eventually, however, the allocation will succeed.

In general, if an instruction reads the value of a spilled temporary, the code is rewritten to read the value from memory into a new temporary just before that instruction. If a value is written to a spilled temporary, the code is rewritten to write the value to a new temporary, followed by an instruction to write the new temporary to memory.

However, AMD64, like x86, allows either the source or the destination of an instruction to be a memory operand. If both operands are spilled, then one of them is rewritten using a new temporary. If only one operand is spilled, then no rewrite occurs and no new temporary is allocated. The frame management pass later converts these spilled temporaries to memory operands in the stack frame.

The main change in register allocation for AMD64 concerns the treatment of floating-point variables. Traditionally, x86 has used the `x87` floating-point unit, which has an 8-entry stack instead of individually accessible registers. This requires additional analysis and transformations for good performance on floating-point intensive programs [11]. AMD64 supports both x87 and the register-oriented SSE2 floating-point unit, with SSE2 being preferred for new code. In HiPE, register allocation for SSE2 uses the same iterated coalescing allocator used for general-purpose registers, but with parameters which are specific for SSE2. HiPE/AMD64 can also target the x87 via a compile-time option, using the same code as in HiPE/x86, but this is mainly intended for testing and benchmarking.

A few minor changes for AMD64 reflect the changes in intermediate representation over x86. Byte-level memory accesses can use any general-purpose register on AMD64, but on x86 we forced them to use %eax. An instruction was added for moving a 64-bit immediate into a register, requiring changes in the code computing def–use information, the code which rewrites instructions when a temporary has been spilled, and the code which applies the final temporary-to-register mapping. The instruction used for indexing and jumping via jump tables was changed to reference the jump table via a register instead of using a 32-bit address constant; this required similar changes as described above.

There are currently three production-quality register allocators available in HiPE/AMD64: one based on linear scan [15, 16], a Briggs-style graph-coloring allocator [4], and an iterated coalescing graph-coloring allocator [7]. The current default is iterated coalescing, but the user can choose between them using a compiler option.

5.3 Frame Management

After register allocation the back-end introduces stack frames and the call stack, maps spilled temporaries to slots in the stack frame, rewrites uses of spilled temporaries as memory operands in the stack frame, creates stack descriptors at call sites, and generates code to allocate, deallocate, and rewrite stack frames at function entry, exit, and at tailcalls. Algorithmically this code is the same as for HiPE/x86 [14, Section 5.3], but it needed many changes to work on AMD64.

The frame module for x86 assumed a 4-byte word size and contained many size/offset constants (4 or 8) based on this assumption. On AMD64, many of these had to be made twice as large. This was done manually since the rôle of each constant had to be checked first.

Eventually both the AMD64 and x86 frame modules were cleaned up to base their calculations on a word size parameter. They are now identical, except for their references to other architecture-specific modules, and for the treatment of floating-point values which occupy two words on x86 but only one word on AMD64. The two implementations could be merged, but we have not done so yet.

5.4 Code Linearization

At this point the symbolic AMD64 code is in its final form, but still represented as a control flow graph. To allow the linearized code to match the static branch prediction rules, we bias conditional branches as unlikely to be taken (if necessary by negating their conditions and exchanging their successor labels). The conversion from CFG to linear code generates the most likely path first, and then appends the code for the less likely paths. Conditional branches in the likely path thus tend to be unlikely to be taken and in the forward direction, which is exactly what we want.

This part of the compiler is identical to that for x86.

5.5 Pseudo-instruction Expansion

As mentioned earlier, calls and conditional branch instructions are pseudo-instructions that list all their possible destinations. After linearization, we rewrite each as a normal AMD64 instruction with no or only one label, followed by an unconditional jump to the fall-through label.

This part of the compiler is identical to that for x86.

5.6 Peephole Optimization

Before assembling the code, peephole optimization is done to perform some final cleanups. It, for instance, removes jumps to the next instruction, which occur as an artifact of the code linearization and pseudo-instruction expansion steps, and also instructions that move a register to itself, which occur as an artifact of the register allocation step.

This part of the compiler is similar to that for x86.

5.7 Assembling

The assembly step converts the symbolic assembly code to binary machine code, and produces a loadable object with the machine code, constant data, a symbol table, and the patches needed to relocate external references.

This is a complex task on AMD64, so it is divided into three distinct passes.

5.7.1 Pass 1: Instruction Translation

The first pass translates instructions from the idealized form used in the back-end to the actual form required by the AMD64 architecture. This is non-trivial:

1. First the types of an instruction's operands (register, memory, small constant, large constant) are identified.

2. Then the set of valid encodings of the operation with those particular operand types is determined.

3. Among the valid encodings, the cheapest (generally the shortest) one should be identified and chosen. This involves knowing about special cases that can use better encodings than the general case. For example, adding a constant to %eax can be done with the standard opcode byte and a byte for the register operand, or with an alternate opcode byte. The size of a constant also matters, since many contexts allow a small constant to be encoded as a single byte, where the general case requires four bytes.

4. While searching for the best encoding of an instruction or its operands, care must be taken to observe any restrictions that may be present. For instance, a memory operand using %rsp or %r12 as a base register *must* use an additional SIB byte in its encoding.

Other instruction-selection optimizations, such as using test instead of cmp when comparing a register against zero, or using xor instead of mov to clear a register, are also worthwhile. In HiPE/AMD64, they are done in the peephole optimization step.

The main change from x86 is the handling of the new SSE2 instructions, most of which are easy to encode. A new case of operands had to be added for the mov instruction, for when an integer register is converted and copied into a floating-point register.

The floating point negation instruction needed major magic. The AMD64 back-end takes an implementation shortcut and represents it as a virtual negation instruction up to this point. The problem is that SSE2 does not have such an instruction. Instead, an `xorpd` instruction must be used to toggle the sign bit in the floating-point representation. Constructing the appropriate bit pattern into another floating-point register at this point would be very awkward. Instead, the bit pattern is stored in a variable in the runtime system, and the `xorpd` gets a memory operand that refers to the address of this variable. However, this memory operand only has 32 bits available for the address. Loading the full 64-bit address into a general-purpose register is out of the question since this runs after the register allocator. Here we are saved by the HiPE/AMD64 code model, which restricts runtime system addresses to the low 32 bits of the address space. In hindsight, it is clear that floating-point negation should have been handled in the pseudo-instruction expansion step instead.

5.7.2 Pass 2: Optimizing branch instructions

Branch instructions have two forms, a short one with an 8-bit offset, and a long one with a 32-bit offset. The shorter one is always preferable, since it reduces code size and improves performance. AMD64 and x86 are identical in this respect.

However, the offset in a branch instruction depends on the sizes of the instructions between the branch and its target, and the sizes of branch instructions in that range may depend on the sizes of other instructions, including the very branch instruction we first considered. This is a classical "chicken-and-egg" problem in assemblers for CISC-style machines, but one that has not received much research attention since the 1970's.

To solve this problem, the HiPE assembler now uses Szymanski's algorithm [17], which is fast and produces optimal code. The algorithm is implemented in a generic module, used by several of HiPE's back-ends.

5.7.3 Pass 3: Instruction encoding

In the last pass the AMD64 instructions are translated from symbolic to binary form. This pass also derives the actual locations of all relocation entries.

In principle, this is straightforward: check each instruction and its operands against the permissible patterns as specified in the AMD64 architecture manuals, select the first that matches, and produce the corresponding sequence of bytes.

The main changes from x86 concern 64-bit operations, the additional registers, and detecting when to emit the new REX prefix.

On x86, encoding an instruction is a simple sequential process: after identifiying the types of the operands, a list is constructed by concatenating the opcode byte(s), the encoding of the operands, and the encoding of any additional immediate operands.

On AMD64, the REX prefix must precede the opcode, but the need to use a REX prefix, and the data to store in it, is not known until later when the operands have been encoded. To handle this we insert partial REX "markers" in the list of bytes when we detect that a REX prefix is needed, for instance if one of the new registers is used. Afterwards, the markers are extracted and removed from the list. If any markers were found, they are combined to a proper REX prefix, which is added at the front of the list. This approach is taken because the compiler is written in Erlang, so it cannot use side-effects to incrementally update a shared "REX needed?" flag.

Both before and after pass 2, it is necessary to know the size of each instruction, in order to construct the mapping from labels to their offsets in the code. For x86, this traverses instructions and their operands just like when encoding them, except it only accumulates the number of bytes needed for the encoding. For AMD64, this does not quite work because of the REX prefixes, so we currently encode the instruction and return the length of that list instead.

Additional changes had to be made to support SSE2 instructions, and to remove the few x86 instructions no longer valid in 64-bit mode, but these changes were straightforward.

In principle the AMD64 encoding module could also be used for x86, if checks are inserted to ensure that no REX prefixes are generated. We have not done so yet, to minimise the risk of adding bugs to the x86 back-end, but it would probably simplify code maintenance.

6. PERFORMANCE EVALUATION

In order to evaluate the performance of the AMD64 port of HiPE, we compare the speedups obtained on this platform with those obtained by the more mature HiPE/SPARC and HiPE/x86 back-ends.

6.1 Performance on a mix of programs

Characteristics of the Erlang programs used as benchmarks in this section are summarized in Figure 3. As can be seen in Figure 4, the speedups of HiPE/AMD64 compared with BEAM are significant (ranging from 35% up to almost 8 times faster). Moreover, more often than not, they are on par or better than those achieved by HiPE/SPARC and HiPE/x86. Compared with the x86, whose back-end is similar to AMD64, the obtained speedups are overall slightly better, most probably due to having double the number of registers. (The only outliers are **tak**, **qsort**, **decode** and **yaws**, for which we currently can offer no explanation.)

The benchmark where the speedup is the smallest is **life**. The obtained speedup is small because this program spends most of its execution time in the process scheduler, which is part of the runtime system of Erlang/OTP (written in C) that is shared across BEAM and HiPE. The benchmark where overall the speedup is biggest, **prettypr**, recurses deeply and creates a stack of significant size. As such, it benefits from generational stack collection [5] which is performed by HiPE's extension to the runtime system (aided by stack descriptors that the HiPE compiler generates), but not when executing BEAM bytecode.

6.2 Performance on programs manipulating binaries

The binary syntax [13] has been an important addition to the Erlang language and nowadays many telecommunication protocols have been specified using it. An efficient compilation scheme of Erlang's bit syntax to native code has been presented in [8].

Speedups on programs manipulating binaries are shown in Figure 5. They show more or less the same picture as Figure 4 with the exception of **descrypt** (a DES encryptor/decryptor), which shows significantly higher speedups running in native code. The reason is that this program

fib A recursive Fibonacci function. Uses integer arithmetic to calculate `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 message. A medium-sized benchmark (≈ 400 lines).

life A concurrent benchmark executing 10,000 generations in Conway's game of life on a 10×10 board where each square is implemented as a process. This benchmark spends most of its time in the scheduler.

yaws An HTML parser from Yaws (Yet another Web server) parsing a small HTML page 100,000 times.

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 ranking 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.

Figure 3: Description of benchmark programs used in Figure 4.

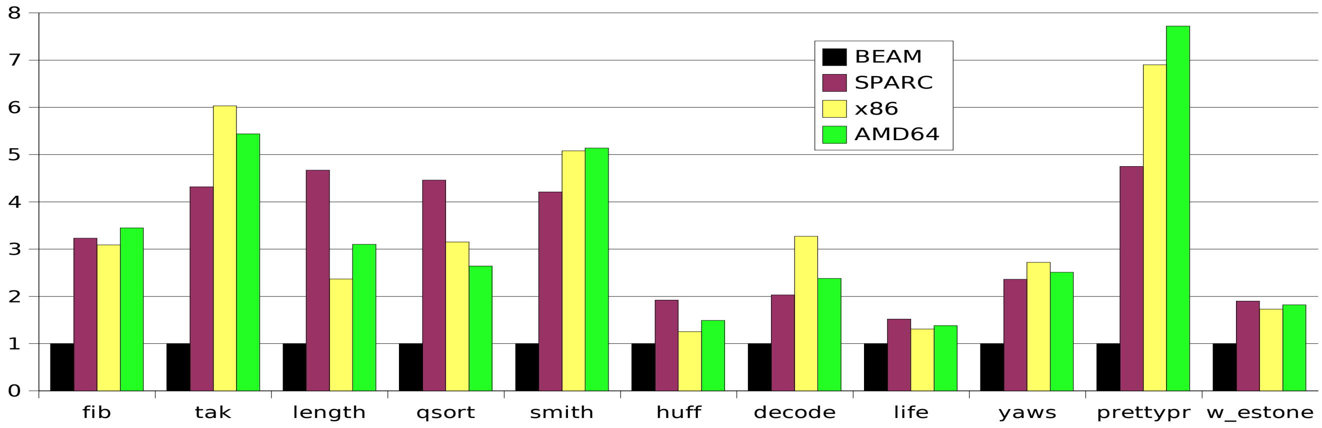

Figure 4: HiPE vs. BEAM speedups across different platforms.

manipulates mostly *bytes* and benefits from HiPE's type analyzer which in this case is able to infer that arithmetic operations on bytes will never overflow or result in a bignum. Because of severe limitations on the number of registers which can be used for byte operations on x86, the obtained speedup is significantly smaller on this platform compared with those on the SPARC and on AMD64.

6.3 Performance on programs manipulating floats

Floats are not extremely common in "typical" Erlang applications, but as the range of uses of the Erlang/OTP system is expanding, they are crucial for some "non-standard" Erlang uses; e.g. for Wings3D. Moreover, the representation of floats differs significantly between a 32-bit and a 64-bit machine. So, we were curious to see the performance of HiPE/AMD64 on floating-point intensive programs.

Figure 6 shows the obtained results. As can be seen, the speedups on all program are better on AMD64 than on x86, due to less memory accesses and having double the number of available FP registers. The **float_bm** benchmark, which requires more than 16 registers, achieves a better speedup on SPARC.

7. AMD64 AS A TARGET MACHINE

Some of the benefits the AMD64 brings over the x86, in general and to Erlang users in particular, include:

- More registers (twice as many as x86) which improves runtime performance by increasing the chance that a compiler will be able to keep an important value in a register as opposed to accessing it in memory.
- Uniform support for byte-level accesses for all integer registers. This improves performance for bit-syntax operations since it eliminates the awkward limitations in the x86.
- Much larger range for fixnums. Since a majority of bit-syntax integer matching operations are for integers less than 60 bits wide, the compiler can generate faster code for these since it knows the results will be representable as fixnums.

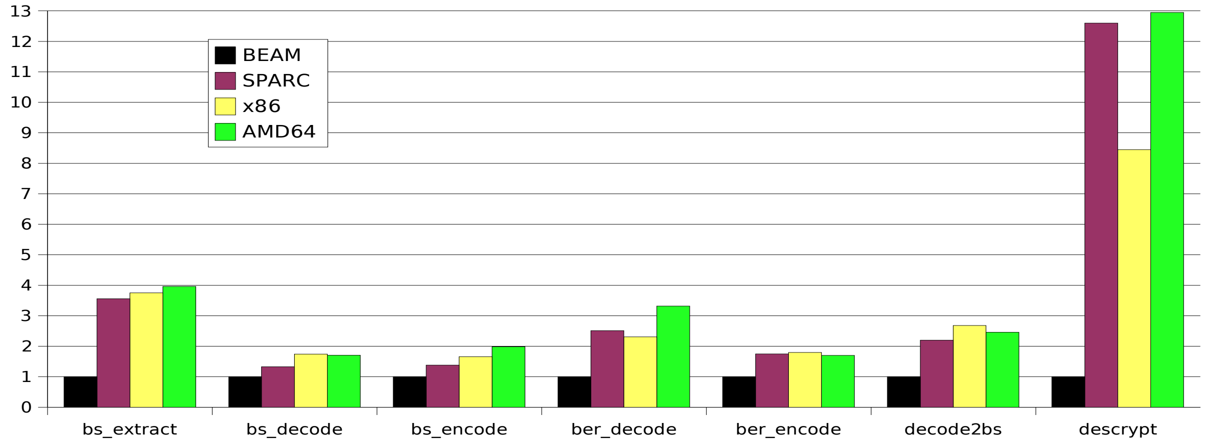

Figure 5: HiPE vs. BEAM speedups on programs manipulating binaries.

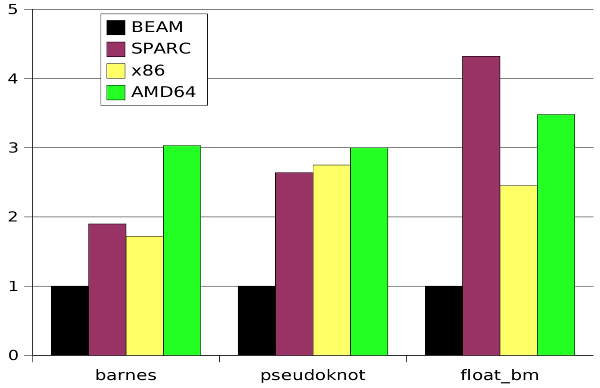

Figure 6: HiPE vs. BEAM speedups on programs manipulating floats.

- Faster operations on big integers. Bignums are represented as sequences of small integers, each half the size of the machine's word size, and arithmetic operations are performed on one such integer at a time. A larger word size means that fewer operations are needed when performing arithmetic on a given bignum.

- Larger address space, both virtual and physical, allowing Erlang to work on e.g. large memory-resident databases.

There is one generic drawback of 64-bit machines over 32-bit machines, viz. that pointer-based data structures such as lists and tuples become twice as large, placing additional burdens on the memory and cache subsystems. It is possible that a "small data" model which restricts pointers and fixnums to 32-bit values may offer the best performance for some applications. We intend to investigate the expected performance benefits of this model.

8. CONCLUDING REMARKS

This paper has described the HiPE/AMD64 compiler: its architecture, design decisions, technical issues that had to be addressed and their implementation. As shown by its performance evaluation, HiPE/AMD64 results in noticeable speedups compared with interpreted code across a range of Erlang programs. Quite often, the obtained speedups are better than those achieved by HiPE/SPARC and HiPE/x86.

HiPE/AMD64, which will be included in the upcoming R10 release of Erlang/OTP system, is the first 64-bit native code compiler for Erlang. However, since 64-bit machines are here to stay, HiPE/AMD64 is most probably not the last compiler of this kind.

9. ACKNOWLEDGMENTS

The development of HiPE/AMD64 has been supported in part by VINNOVA through the ASTEC (Advanced Software Technology) competence center as part of a project in cooperation with Ericsson and T-Mobile.

10. REFERENCES

[1] AMD Corporation. *AMD64 Architecture Programmer's Manual*, Sept. 2003. Publication # 24592, 24593, 24594, 26568, 26569.

[2] AMD Corporation. *Software Optimization Guide for AMD Athlon^{TM} 64 and AMD Opteron^{TM} 64 Processors*, Sept. 2003. Publication # 25112, Revision 3.03.

[3] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang.* Prentice Hall Europe, Herfordshire, Great Britain, second edition, 1996.

[4] 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.

[5] 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, New York, N.Y., 1998. ACM Press.

[6] 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.

[7] L. George and A. W. Appel. Iterated register coalescing. *ACM Trans. Prog. Lang. Syst.*, 18(3):300–324, May 1996.

[8] P. Gustafsson and K. Sagonas. Native code compilation of Erlang's bit syntax. In *Proceedings of ACM SIGPLAN Erlang Workshop*, pages 6–15. ACM Press, Nov. 2002.

[9] 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, New York, NY, Sept. 2000. ACM Press.

[10] 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*, 4(4):421–436, Aug. 2003.

[11] T. Lindahl and K. Sagonas. Unboxed compilation of floating point arithmetic in a dynamically typed language environment. In R. Peña and T. Arts, editors, *Implementation of Functional Languages: Proceedings of the 14th International Workshop*, number 2670 in LNCS, pages 134–149. Springer, Sept. 2002.

[12] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufman Publishers, San Fransisco, CA, 1997.

[13] P. Nyblom. The bit syntax - the released version. In *Proceedings of the Sixth International Erlang/OTP User Conference*, Oct. 2000. Available at http://www.erlang.se/euc/00/.

[14] M. Pettersson, K. Sagonas, and E. Johansson. The HiPE/x86 Erlang compiler: System description and performance evaluation. In Z. Hu and M. Rodríguez-Artalejo, editors, *Proceedings of the Sixth International Symposium on Functional and Logic Programming*, number 2441 in LNCS, pages 228–244, Berlin, Germany, Sept. 2002. Springer.

[15] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Trans. Prog. Lang. Syst.*, 21(5):895–913, Sept. 1999.

[16] K. Sagonas and E. Stenman. Experimental evaluation and improvements to linear scan register allocation. *Software – Practice and Experience*, 33(11):1003–1034, Sept. 2003.

[17] T. G. Szymanski. Assembling code for machines with span-dependent instructions. *Communications of the ACM*, 21(4):300–308, Apr. 1978.

[18] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. Prog. Lang. Syst.*, 13(2):181–210, Apr. 1991.