Uppsala Master's Theses in
Computing Science 286
Examensarbete DV3
2004-12-24
ISSN 1100–1836

# An AMD64 Backend for HiPE: Implementation, Performance Evaluation, and Lessons Learned

**Daniel Luna**
luna@update.uu.se

**Information Technology**
**Computing Science Department**
**Uppsala University**
**Box 337**
**S-751 05 Uppsala**
**Sweden**

## Abstract

This master's thesis describes the construction of the AMD64 backend for the HiPE (High Performance Erlang) native code compiler, a part of Erlang's primary implementation, the Erlang/OTP. More specifically it describes the work done when writing the compiler backend, some implementation choices that had to be made, and the performance effect of these choices.

This thesis consists of two papers; the first describes the internals of the compiler backend, technical issues that had to be addressed in its development, and reports on its performance compared to the older HiPE backends and interpreted Erlang code. The second paper takes a deeper look at AMD64 backends in general, examines a few different implementation options and reports on their performance.

Supervisor & Reviewer: Konstantinos Sagonas kostis@it.uu.se

Passed:

# 1 Introduction

Modern computers are equipped with more memory than before and can therefore run programs that are bigger and more memory demanding. Even today there are programs that reach the memory limit for 32 bit machines. Unfortunately a 32 bit register only allows $2^{32}$ different values, thus limiting the available address space to 4 GB. With 64 bit registers this problem is pushed well into the future.

AMD64 from AMD is a 64 bit processor family[1] that also allows already existing 32 bit x86 programs to be run side-by-side with newer 64 bit programs. This advantage, together with the affordability of the architecture, is what made us choose AMD64 as the target for a 64 bit backend for HiPE.

HiPE (High Performance Erlang) is a native code compiler for Erlang, a concurrent functional programming language designed for developing large-scale, distributed, fault-tolerant systems. The primary implementation of Erlang, the Erlang/OTP, is by default based on a virtual machine interpreter (BEAM); but with HiPE also allows the user to natively compile those parts of the code where the speedups are worth the larger code size and longer compilation times, and keep the non-time-critical part of the application in interpreted code (i.e bytecode).

The HiPE compiler is currenly available for the x86, SPARC V8+, PowerPC and – since recently – AMD64 architectures. This master's thesis presents the work of creating the AMD64 backend of HiPE. It consists of two papers that each describe different parts of that process and how other compiler writers might benefit from our experience. This introductory text summarizes these two papers and also tries to present ideas on future improvements for HiPE/AMD64.

The work presented here has been part of Erlang/OTP since R10B-0[2].

# 2 "HiPE on AMD64"

The first paper[1] describes the porting of HiPE to the AMD64 architecture and its target readership are developers from the Erlang community. It discusses technical issues that had to be addressed during the development of the port, and reports on the speedups compared with BEAM.

While the paper includes a brief overview of the AMD64 architecture and HiPE's rôle in Erlang/OTP, the main contributions are the new compiler backend and the changes to the runtime system. Some of the efforts spent on the project was also aimed at making HiPE and Erlang/OTP 64 bit clean, something that was believed to be true at the start of the project. This problem manifested all through the work on the different parts of the AMD64 backend.

The paper describes how the AMD64 runtime system was created from one existing for x86. For instance, the creation of code stubs for interpreted Erlang functions had to be rewritten to allow for a 64 bit address space, while the code for signal handling could be kept as it was. At the end, about a third of the code could be shared by the x86 and the AMD64 runtime systems. These changes also resulted in a cleaner runtime system with higher performance on 64 bit machines.

[1] Intel's server architecture EM64T is binary compatible with AMD64.

[2] Available at http://www.erlang.org/.

A detailed description of the AMD64 backend appears in section 5 of the paper. While the changes to allow for more registers were straightforward, other problems were more difficult to address. For instance, AMD64 has inherited from x86 a 32 bit upper limit on the size of immediates, with the sole exception of the new `mov reg imm64` instruction. This forces the use of an extra register whenever a 64 bit constant is needed. This happens, for instance, with references to jump tables. This section also explains why SSE2, instead of x87, was chosen for the floating point processor support, and furthermore goes into detail on the construction of the assembler.

In the performance evaluation it can be seen that the newly developed HiPE/AMD64, when compared to interpreted code, has noticeable speedups across a range of Erlang programs. These speedups are comparable to, and often better than, the ones for the more mature x86 and SPARC backends.

# 3 "Super Size your Backend: Advice on how to Develop an Efficient AMD64 Backend"

The second paper[2] focuses on how other compiler writers might benefit from the work done on HiPE/AMD64. It describes how an AMD64 compiler backend can be created from one for x86, and offers advice based on our experiences. This paper describes issues related to developing an efficient AMD64 backend in general and uses HiPE/AMD64 as an example of this process.

In the performance evaluation it can be seen that compared with 32 bit mode the 64 bit mode is, in general, faster. Unfortunately 64 bit programs are larger, but this is almost always a price worth paying.

Since HiPE has the advantage of having four different register allocators the paper also tries to give advice on which to choose. With the greater amount of registers for the AMD64, the choice of allocator is not as important as for the x86, and the quick answer is that for most applications the linear scan allocator described in [3] achieves almost as good results as the more complex graph coloring allocators.

For the benchmarks in the paper it is unclear whether there is any gain in reserving registers for parameter passing, but for some applications this is very useful, so the answer must be *yes*. The performance evaluation ends with a comparison between the SSE2 and x87 floating point units, and a note on whether to use the native stack or simulate it.

The AMD64 platform offers an opportunity for compiler writers to compile for an additional target with moderate effort and, since most compilers already have an x86 32 bit backend, is also a good choice for any compiler's first 64 bit backend.

# 4 The Future and Conclusions

Writing this compiler backend has been fun and, in some sense, successful. With the HiPE compiler's new backend, the advantages of 64-bit architectures are available to Erlang developers. There are some future improvements that could be done, though.

Since the differences between the x86 and AMD64 HiPE backends are small, it is possible to merge many of the files. Using macros, large parts of the AMD64 compiler backend could be changed into a few `include` statements and some glue code. This would help to keep the backends in sync and decrease the burden of their maintainance.

Another possible improvement is to implement support for conditional assignments. Since AMD64 is a rather new architecture, all AMD64 compliant processors support them. Some parts of the compiler (most notably the code generated for large `case` statements) would benefit from using conditional assignments instead of jumps, thus allowing for better hardware branch prediction. This, and other processor-specific optimizations, is a project for the future.

Also worth noting is that when HiPE/x86 got its floating point support a few years ago, not all x86 processors had SSE2 support and a design decision was made to use the x87 stack instead. With the newer AMD64 architecture all processors have support for these instructions and HiPE/AMD64 uses SSE2 by default. This option could be backported into the x86 backend.

Finally, I want to note that the development of HiPE/AMD64 has been a very valuable and rewarding experience.

# References


[1] D. Luna, M. Pettersson, and K. Sagonas. HiPE on AMD64. In *Proceedings of the 2004 ACM SIGPLAN workshop on Erlang*, pages 38–47. ACM Press, 2004.

[2] D. Luna, M. Pettersson, and K. Sagonas. Super size your backend: Advice on how to develop an efficient AMD64 backend. Oct. 2004.

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

# 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.


[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.

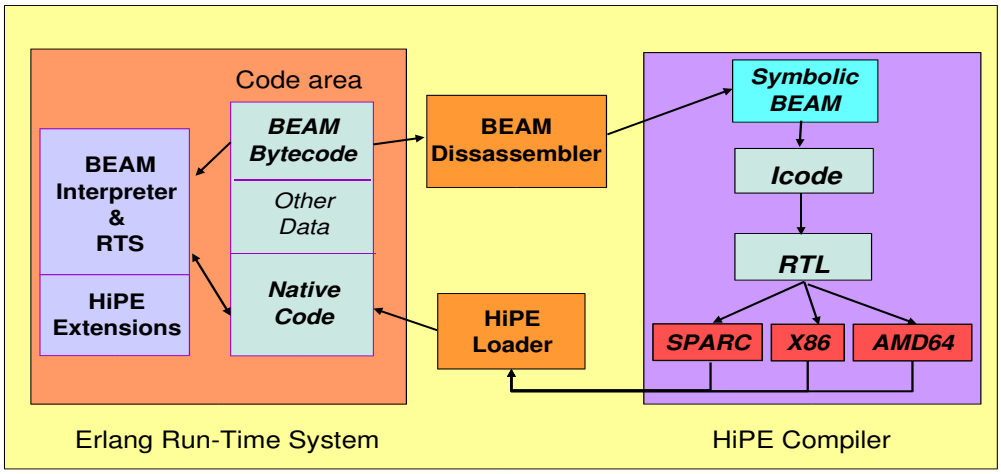



**Figure 1: Architecture of a HiPE-enabled Erlang/OTP system.**

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].

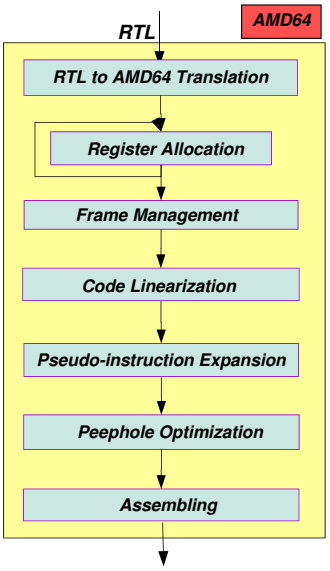



**Figure 2: The AMD64 back-end of Fig. 1 in detail.**

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 ($\approx 400$ lines).

**life** A concurrent benchmark executing 10,000 generations in Conway's game of life on a $10 \times 10$ 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.**

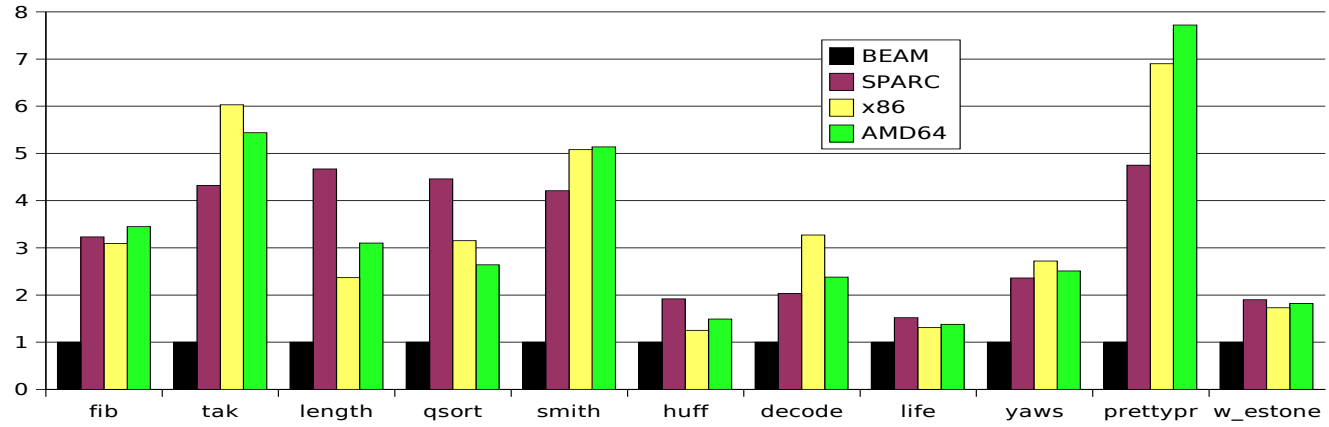



**Figure 4: HiPE vs. BEAM speedups across different platforms.**

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.

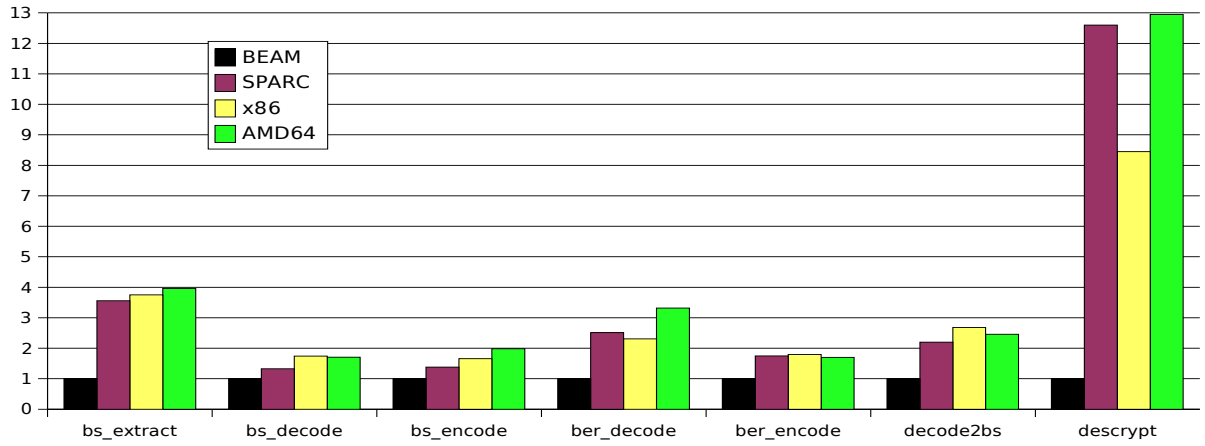



**Figure 5: HiPE vs. BEAM speedups on programs manipulating binaries.**

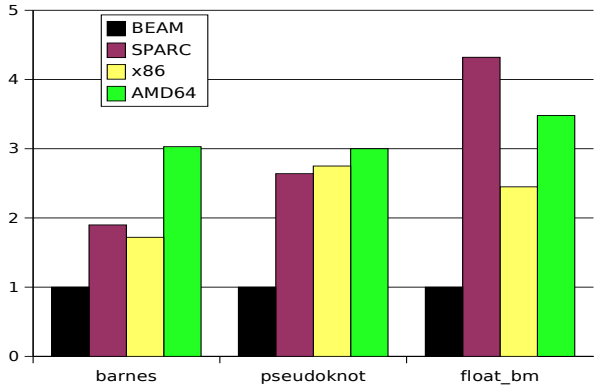



**Figure 6: HiPE vs. BEAM speedups on programs manipulating floats.**

- 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.

# Super Size your Backend: Advice on how to Develop an Efficient AMD64 Backend

Daniel Luna, Mikael Pettersson, and Konstantinos Sagonas

Department of Information Technology, Uppsala University, Sweden
e-mail: luna@update.uu.se {mikpe,kostis}@it.uu.se

**Abstract.** For about a year now, we have been developing and tuning an AMD64 backend for the HiPE Erlang compiler. In this paper, we try to step back and critically examine design choices for obtaining an efficient AMD64 backend. We describe how other developers can migrate existing x86 backends to the AMD64 architecture, and offer advice based on our experiences. In particular, we mention backend components that can be shared between x86 and AMD64, and those that better be different for achieving high performance on AMD64. Finally, we measure the performance of several different alternatives in the hope that this can save development effort for others who intend to engage in a similar feat.

## 1 Introduction

It is hardly surprising that developing an efficient new backend for an existing compiler, especially in compilers for high-level languages, usually turns out to be a bigger task than one initially anticipates. To do it properly, one should ideally consider many alternatives for each design choice and experimentally evaluate their performance tradeoffs. As this is much more easily wished than done, few compiler teams actually invest this effort and follow this ideal approach to backend development. We hold that the effort spent in developing a backend for a new architecture can be reduced significantly if experiences get documented on paper and shared among compiler writers. This way, developers can readily choose parameters that worked well in settings which are similar to theirs, and only vary design choices that rely on assumptions which are not valid in their compiler framework.

In the context of the HiPE compiler,[1] an industrial-strength native code compiler for the concurrent functional language Erlang, we have spent the period November 2003 – September 2004 developing and tuning a backend for AMD64. (Actually, this has been the first 64-bit backend of HiPE. The AMD64 platform was chosen due to its similarities with the widely popular x86 architecture, its upcoming importance,[2] and the affordability of these machines.) We have experimented with various implementation alternatives and, since the implementation issues at the level of generating native code are generic, we believe that our experiences and measurements are of interest to all compiler writers that consider developing an efficient AMD64 backend.

The contributions of this paper are:

- On the 'theoretical' side, the paper offers advice on how to develop an efficient compiler backend for AMD64. In particular, we describe how to migrate an existing backend for

[1] The High Performance Erlang compiler; see www.it.uu.se/research/group/hipe/.

[2] In addition to many major PC manufacturers that already provide AMD64-based desktops and laptops, Sun has announced its new line of AMD Opteron 64 based servers and workstations; see www.sun.com/amd/.

x86 to AMD64 with moderate effort, but without missing opportunities to take advantage of architectural features present on AMD64 but not on x86.
- On the experimental side, the paper includes an extensive set of measurements (obtained using hardware performance counters) that evaluate the performance of various implementation alternatives and support our advice.

As a by-product, this paper also documents the internals of HiPE/AMD64 and compares our choices with those of the few other compilers with existing AMD64 backends.

The rest of the paper is structured as follows. The next section describes the infrastructure of the HiPE compiler, while Sect. 3 overviews the characteristics of the AMD64 architecture from a compiler writer's perspective. Sections 4 and 5 form the main body of this paper describing in detail issues that are generic to obtaining good performance on AMD64 and issues that are particular to the implementation of functional languages, respectively. The performance of all these implementation alternatives is evaluated in Sect. 6. The paper ends with reviewing the internals of other currently existing compilers with backends for AMD64 (Sect. 7) and with some concluding remarks (Sect. 8).

## 2 The HiPE Compiler Infrastructure

In this section, we briefly describe the Erlang/OTP system and the HiPE native code compiler which is the basis of our work; refer to [9, 13] for more detailed information.

Erlang is a concurrent functional language designed for developing large-scale, distributed, fault-tolerant, soft real-time control systems such as those typically developed by the telecom industry. The primary implementation of the language is the Erlang/OTP system from Ericsson, which is nowadays used by many big companies to develop large (million lines of code) commercial applications. Even though Erlang/OTP is by default based on a virtual machine interpreter, since 2001, it also includes the HiPE native code compiler as a fully integrated component.[3]

The HiPE compiler currently has backends for SPARC V8+, x86, AMD64, and PowerPC. It can be used as either a just-in-time or ahead-of-time compiler, and compilation can start either from bytecode or from source. (However, in this paper, all measurements were obtained in the mode where compilation happens ahead-of-time and starts from bytecode.) The target-independent part of the compilation takes place in two intermediate code representations: Icode and RTL.

Icode is internally represented as a *control flow graph* (CFG) which has been turned into static single assignment (SSA) form [4]. In this stage various optimizations are performed: sparse conditional constant propagation (SCCP) [18], unreachable and dead-code elimination (DCE), and copy propagation (CP). Finally, a type propagator guided by dataflow analysis 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. In RTL, almost all operations are made explicit. For example, data tagging and untagging is translated to appropriate machine

[3] Erlang/OTP exists both in commercial and open-source distributions and their differences are mostly in support; see `www.erlang.se` and `www.erlang.org`, respectively. Since October 2004, the HiPE/AMD64 compiler is part of Erlang/OTP R10B and can be obtained from there.

operations (`shift`, `or`, *etc*), data accesses are turned into loads and stores. Also arithmetic operations, data constructions and type tests are inlined. RTL is also internally represented as a CFG in SSA form and similar optimizations as in Icode (SCCP, DCE, and CP) as well as partial redundancy elimination (PRE) are performed.

Finally, RTL code is translated to (a symbolic representation of) the language of the target backend. At this level the most important compilation phases that take place are register allocation, branch-prediction-aware trace linearization, some peephole optimizations, and finally assembling.

As far as this paper is concerned, it is important to notice that at the levels of RTL and machine language, issues are independent from the choice and characteristics of language from which HiPE starts its compilation. The only major 'typical' compiler optimizations that the HiPE compiler currently does not perform are loop optimizations and instruction scheduling. The former is not so effective in a functional language where functions are not nested. The latter is of limited importance for an architecture which performs out-of-order execution of instructions, such as AMD64.

## 3 An Overview of AMD64

AMD64 is a family of general-purpose processors currently available 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 characteristic is that they are fully compatible with 32-bit x86 code. AMD64 processors can run 32-bit operating systems and applications (referred to as *legacy mode*), 64-bit operating systems and applications, or 64-bit operating systems with 32-bit applications (called *compatibility mode*).

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 since started making software-compatible 64-bit processors, initially for servers.[4]

### 3.1 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 are optional.
- 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.

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

– 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. This means that an efficient compiler should pay attention to how it constructs traces and linearizes its code.
– 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 to fifteen bytes. The actual instruction 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.

### 3.2 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. 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 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.

Immediate operands on AMD64, 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. 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.

The 32-bit immediate operands on AMD64 are zero-extended when used in 32-bit operations (the default), but sign-extended when used in 64-bit operations. This makes it difficult to use constants and addresses in the $[2^{31}, 2^{32} - 1]$ range in 64-bit operations.

x86 has several ways of encoding a memory operand that denotes an absolute 32-bit address. AMD64 redefines one of those encodings to instead denote a PC-relative address. This is particularly helpful for reducing the number of load-time relocations in programs with many accesses to global data.

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.

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. Although newer x86 processors also implement SSE2, they are limited to 8 registers; furthermore, unless told otherwise, a compiler for x86 cannot assume that SSE2 is available.

# 4 Language-Independent Efficiency Considerations

For generation of efficient code on AMD64, there are a few general but important rules to obey [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. Avoid unnecessary REX prefixes.
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.

## 4.1 Immediate Operands

Immediate values (constants) in general operands are limited to 32 bits on AMD64, as on x86. On AMD64, an immediate is sign-extended to 64 bits when used in a 64-bit operation, while 32-bit operations compute 32-bit results which are then zero-extended to 64 bits before being stored in a target register.

An obvious consequence of this is that instructions containing large immediates may have to be rewritten on AMD64. If a constant in the $[2^{31}, 2^{32} - 1]$ range is to be simply loaded into a register or stored in a 32-bit memory word, then there is no problem because a 32-bit operation will have the desired effect: if the target is a register then the result is zero-extended to 64 bits, and if the target is a memory operand, the result is truncated to 32 bits. On the other hand, if such a constant is to be used as an operand in a 64-bit operation, like an addition or a 64-bit memory write, then the code must be modified to compute the constant into a register first, and to use that register instead of the constant in the original instruction.

Another consequence is that code or data at arbitrary 64-bit addresses cannot be accessed using only immediate operands: in general, 64-bit addresses must be loaded into registers which are then used to access the code or data indirectly.[5] A *code model* is a set of constraints on the size and placement of code and data, the idea being that runtime overheads can be reduced by sacrificing some generality. The C ABI document for AMD64 [8] defines the following three basic code models for application code:[6]

**Small code model** All compile-time and link-time addresses and symbols are assumed to fit in 32-bit immediate operands. This model avoids all overheads for large addresses, but restricts code and global data to the low 2GB of the address space, due to sign-extension of immediate operands.

[5] This is a generic issue on machines which only have immediate operands smaller than their virtual address space. The issue also affects 32-bit SPARC and PowerPC, but not x86.

[6] The ABI also defines code models for position-independent code and the Linux kernel.

**Medium code model** Like the small code model, except that addresses of global data are unrestricted. To construct a large address, the compiler must use a new form of the `move` instruction which loads a 64-bit immediate constant into a register. Calls and jumps to code can still use ordinary 32-bit immediate offsets.

**Large code model** No restrictions are placed on the size or placement of either code or global data. Global data accesses are as in the medium code model. Long-distance calls and jumps must use indirection: this can be done statically, by rewriting all call and jumps that *may* be long-distance, or dynamically, by having the linker or loader redirect long-distance calls and jumps to automatically generated *trampolines* that then jump indirectly to the final targets.[7]

HiPE/AMD64 uses a hybrid small/medium code model. Addresses of code and runtime system symbols are assumed to fit in sign-extended 32 bit immediates. The addresses of data objects defined in compiled code, i.e., compile-time literals and jump tables, are not assumed to fit in 32 bits; for them the `move reg,imm64` instruction is used, which the code loader updates with the datum's actual runtime address.

GCC on AMD64 implements the small and medium code models, with the small one being the default [7]. It also implements a variant of the small code model where all constant addresses are in the last $2^{31}$ bytes of the 64-bit address space; this code model is used for the Linux kernel.

### 4.2 Floating-Point Arithmetic

Floating-point arithmetic on x86 is traditionally done with the old `x87` instruction set, which uses an 8-entry stack. A newer register-oriented instruction set, SSE2, was added in the Pentium 4 processor, but a compiler for x86 cannot utilize it unless it can be sure that the generated code will only run on SSE2-capable processors. AMD64 changes the situation in two ways: SSE2 is guaranteed to be present, and the number of floating-point registers has been doubled to 16.

On AMD64, SSE2 is generally preferred over `x87`, because of the larger number of registers, and because it avoids the complicated analyses and code generation algorithms needed to work around the limitations of the `x87` stack [11]. Although AMD64 processors still support `x87`, there are indications that some operating systems, including Windows, will drop `x87` support when they migrate from 32 to 64 bits.

### 4.3 Register Allocation

Generating efficient code for x86 can be difficult, mainly because the small number of general-purpose registers (7, not counting the stack pointer) results in a larger number of spills than on typical RISC machines. Generating efficient code requires using both computationally intensive register allocation methods, such as graph coloring, and `x86`-specific solutions such as using explicit memory operands instead of reloading spilled temporaries into registers (which might cause other temporaries to spill).

AMD64 considerably improves the situation. The availability of 15 general-purpose registers (excluding the stack pointer) reduces register pressure and the number of spills. This may allow less computationally intensive strategies for register allocation, such as linear scan [14, 15], to become feasible on AMD64; this is especially important when compile times are an issue, such as in JITs and in interactive systems.

[7] HiPE uses trampolines on PowerPC to compensate for its small unconditional branch offsets.

8-bit operations can be awkward on x86 because it only allows the first four general-purpose registers to be used for 8-bit operands. AMD64 allows any general-purpose register to be used for 8-bit operations. Working around the limitations on x86 constrains either instruction selection or register allocation, which can result in performance losses[8]. On AMD64 these constraints are not necessary, allowing the compiler to generate potentially higher-performance code.

### 4.4 Parameter Passing

Passing function parameters in registers is an important optimization in many programming language implementations. First, it tends to reduce the number of memory accesses needed to set up the parameters in the caller. Second, it allows the callee to decide whether the parameters need to be saved on the stack or not. For leaf functions, the parameters can typically remain in registers throughout the function's body. For non-leaf functions, the compiler is free to decide if and where the parameters should be saved on the stack.

Since AMD64 has twice as many general-purpose registers as x86 has, a compiler will in general be able to pass more parameters in registers on AMD64 than on x86, which should improve performance.

An important issue to consider is whether the calling convention needs to be compatible with the standard C calling convention or not. In the former case, the compiler has little choice but to follow the standard rules, which for Unix and Linux are: on x86 all parameters are passed on the stack, on AMD64 the first six are passed in registers with the remainder on the stack just as for x86. If this is the case, then the compiler must be generalized to support register parameters when migrating from x86 to AMD64. In the latter case, the compiler is probably already passing some parameters in registers in x86, so migrating to AMD64 just involves changing the number of parameter registers used, and their names.

### 4.5 Branch Prediction

Enabling good branch prediction is essential for performance for typical integer code, due to such code having a higher degree of tests and conditional branches than typical numerical code. The processor's dynamic branch prediction table takes care of this for the most frequently executed (hot) code, but it cannot do so for infrequently executed (cold) code, or when the amount of hot code is too large for the table.

If the compiler has reason to assume that a given conditional branch is more likely to branch in a particular direction, then it *should* linearize the code so that this prediction coincides with the processor's *static branch predictor*. Modern AMD64 and x86 processors predict forward conditional branches as not taken, and backward conditional branches as taken, so a way to achieve this is to:

1. bias conditional branches to be unlikely to be taken, if necessary by inverting their conditions, and
2. linearize the control flow graph by constructing traces that include the most likely path first.

Assumptions about branch directions may come from a variety of sources, including programmer annotations[9] and feedback from running the code in profiling mode. Dynamically

[8] In HiPE/x86, these limitations are currently worked around by always using the `%eax` register in byte memory accesses.

[9] Such as the `__builtin_expect` annotation in `gcc`.

typed languages typically perform frequent type tests that check for error conditions before primitive operations; these tests can be assumed to be highly biased in the non-error direction.

In a large code model, (potentially) long-distance calls and jumps must use indirection via computed addresses. The targets of such instructions will not be predictable unless the instructions occur in hot code paths. Using normal (static) calls or jumps to trampolines may improve branch predictability by reducing the number of distinct indirect jumps that need to be resolved and recorded in the dynamic branch prediction table.

Since `call` and `ret` instructions push and pop (respectively) return addresses on the return stack branch prediction buffer, it is important to use them in pairs. Not doing so, by for instance manually pushing a return address on the stack but returning to it with `ret`, will cause the buffer to become unsynchronized with the actual stack, which in turn causes branch prediction misses in the `ret` instructions. This issue highly relevant for languages that implement proper tail-recursion optimization; see Sect. 5.1.

### 4.6 Instruction Operand Encoding

x86 encodes instruction operands using so-called ModRM and SIB bytes, which contain modifiers and register numbers. Some combinations of modifiers and register numbers change the interpretation of an operand, leading to a number of special cases which must be handled. The AMD64 REX prefix provides an additional bit for each of the three register number fields in the ModRM and SIB bytes, which affects the rules for the special cases. The updated rules for the existing special cases are:

- A memory operand with a base register can be described with just a ModRM byte, except when the register is `%esp`, in which case an additional SIB byte is required.
  AMD64 does not decode the REX B bit to determine this case. Therefore, a SIB byte is also required when register 12 (`%esp` + 8) is used as a base register.
- A memory operand with a base register but no offset (implicitly zero) can be decribed with just a ModRM byte, except when the register is `%ebp`, in which case an explicit offset constant must be included.
  AMD64 does not decode the REX B bit to determine this case. Therefore, an explicit offset is also required when register 13 (`%ebp` + 8) is used as a base register.
- `%esp` cannot be used as an index register in a memory operand, since that SIB encoding instead indicates the absence of an index.
  AMD64 *does* decode the REX X bit to determine this case. Therefore, there is no problem using register 12 (`%esp` + 8) as an index register.
- A memory operand with a base register and an optional index but no offset can be described with a ModRM and a SIB byte, except when the register is `%ebp`, in which case an explicit offset must be included.
  AMD64 does not decode the REX B bit to determine this case. Therefore, an explicit offset is also required when register 13 (`%ebp` + 8) is used as a base register with an optional index.

AMD64 also adds a new special case. A memory operand specified simply by a 32-bit constant can be encoded in several different ways. AMD64 has redefined the shortest encoding so that the constant is added to the program counter instead of being an absolute address. This is a highly desirable feature since it can be used to reduce the number of load-time relocations, but it forces operands with absolute addresses to use a longer encoding on AMD64 than on x86.

### 4.7 REX Prefixes: Detecting them, Avoiding them

The REX prefix on AMD64 has two uses: it provides additional bits to register numbers in instruction operands, and it provides a flag which switches an instruction from the default 32 bit operand size to a 64 bit operand size.

Detecting the need for a REX prefix is easily done while the assembler is encoding an instruction: any use of a high register number (8–15) or a 64-bit operation triggers it.

On the other hand, REX prefixes increase code size, reducing instruction decode bandwith and the capacity of the instruction cache, so the recommendation [2] is to avoid unnecessary REX prefixes. This can be done by avoiding 64-bit operations when 32-bit ones suffice, and by preferring low register numbers (0–7) over high ones in 32-bit operations. The applicability of these strategies are application and language specific. For instance, most C code uses plain `int` for integers, which are 32 bits on AMD64, while code using pointers or pointer-sized integers (which is typical in high-level symbolic languages) must use 64-bit operations.

## 5 Considerations for Functional Languages

### 5.1 Tail Recursion and Branch Prediction

Functional programming languages typically require proper tail-recursion optimization in their implementations; this is because they omit imperative-style looping statements, leaving tail-recursive function calls as the only way to construct loops. Logic programming languages are similar in this respect.

Consider a call chain where $f$ recursively calls $g$ which tailcalls $h$. $f$ sets up a parameter area including a return address back to $f$ and then branches to $g$. $g$ then rewrites this area and branches to $h$. In $h$, the area *must* look exactly as if $f$ had called $h$ directly. The format and size of the parameter area depends on the number of parameters; a tailcall where the caller and callee have different number of parameters must therefore change the format of the area. Now consider the return address parameter. It will not change at a tailcall, but depending on the formatting rules for the parameter area, it may still have to be moved to a different location. This relocation is pure overhead, so many implementations have focused on avoiding it.

One way to avoid relocating the return address is to always pass it in a specific register; to return, a jump via that register is executed [17]. Another approach is to push the return address on the stack *before* pushing the remaining actual parameters [16, 5]. This ensures that even if caller and callee at a tailcall have different number of parameters, the location of the return address will remain the same. To return, a native return instruction which pops the address off the stack may be used, or the address can be popped explicitly and jumped to via a register. Both approaches have been used to implement tailcalls on stack-oriented machines like x86 and older CISCs[10]. The problem with these approaches is that they cause branch prediction misses at returns, because the return stack branch predictor either is not used at all or is out of sync.

The approach taken in HiPE, on both x86 and AMD64, is to use the native call stack in the natural way. At a recursive call, the parameters are placed in registers or at the bottom of the stack, and the callee is invoked with a `call` instruction. At a return, a `ret $`$n$ instruction is executed which pops the return address and $n$ bytes of parameters and then returns. The

[10] Passing the return address in a register is the normal case for RISCs.

main advantage of this approach is that it enables the return stack branch predictor, which reduces the number of branch mispredictions. It also reduces the number of instructions needed at calls and returns. The only disadvantage is that the return address will have to be relocated at tailcalls if the caller and callee have different number of parameters *on the stack*. However, with sufficiently many parameters passed in registers, the need for relocating the return address becomes less likely. Since AMD64 has more registers available for parameter passing than x86, a calling convention that avoids return address relocation in most cases is quite feasible.

### 5.2 Caching Global State in Registers

Compiled code from functional languages often reference a number of global variables, typically including at least a stack pointer and a heap pointer (for dynamic memory allocation), and usually also stack and heap limit pointers (for memory overflow checking). Erlang, being a concurrent language, adds to these a simulated clock and a pointer to the current process' permanent state record. Having these global variables permanently in registers should in general improve performance. Thanks to its larger number of registers, up to about 4–6 global variables in registers should be possible on AMD64.

Increasing the number of global variables in registers also increases the cost when these registers must be saved to or restored from memory cells. One such case is when the code needs to call procedures written in other languages, such as C library procedures. Another case is context switching for process scheduling in concurrent languages that implement their own processes. In Erlang/OTP, both cases are very frequent.

### 5.3 Native Stack Pointer or Not?

The stack pointer needs additional consideration. As described previously, using the hardware-supported stack in the natural way has advantages for branch prediction and instruction counts; it also avoids reserving a general-purpose register for a rôle directly supported by the hardware stack pointer. Unfortunately, using the hardware stack also has some disadvantages:

- On both AMD64 and x86, a memory operand consisting of a base register and an offset requires a one byte longer encoding when the base register is the hardware stack pointer. This slightly increases the code size for stack accesses.
  As long as reasonable quality register allocation is performed for local variables, it is doubtful that this code size increase is a serious issue.[11] If it does turn out to be an issue, then another register can be reserved, and used either as a frame pointer in addition to the stack pointer, or as a replacement for the stack pointer. Of course, both choices entail performance losses in other areas.
- Some operating systems can force a process to *asynchronously* execute a call to some code on the current hardware stack. This issue arises from signal handlers in Unix and Linux, but it also affects Windows and possibly other operating systems. If the stack is dynamically sized and explicitly managed by the compiled code from the functional language, then the stack may overflow as the result of such an asynchronous call.

[11] MLton was designed to avoid the issue on x86, by using `%ebp` as a pointer to a simulated stack, and reassigning `%esp` to be the heap pointer. There is no benchmark data available measuring the impact of this design choice.

On Unix and Linux, it is possible to force signal handlers to execute on a separate stack, via the `sigaltstack` system call and by registering signal handlers with the `SA_ONSTACK` flag. HiPE, MLton, and Poly/ML all use this solution. No such workaround appears to be possible for Windows, so there the options seem limited to either include a scratch area at the bottom of the stack (the solution used by Poly/ML), or to abandon using the native stack at all (the solution used by MLton).

– Synchronous calls to code written in some other language, such as C library routines, are also susceptible to stack overflow if the stack is dynamically allocated and explicitly managed. If this is the case, then those calls should be implemented such that a *stack switch* is performed to the standard C stack before the call, followed by a switch back afterwards.

  After the stack switch the actual parameters must also be adjusted if the parameter passing conventions differ between the functional language and C. Passing most parameters in registers reduces this cost, even more so on AMD64 than x86 since C on AMD64 takes up to six integer parameters in registers.

  HiPE on AMD64 passes parameters in the same registers as C, which avoids having to copy the parameters at (its frequent) calls to C procedures. On x86, HiPE passes parameters in registers which are then simply pushed on the C stack before a call to C; this is cheaper than copying them from memory cells on the Erlang stack.

## 6 Performance Evaluation

The performance evaluation was conducted on a desktop machine with a 2GHz Athlon64 processor, 1GB of RAM and 1MB of L2 cache, running Fedora Core 2 Linux in 64-bit mode. Measurements for x86 code were obtained by running that code in *compatibility* mode on the same machine. The Linux kernel has been updated with the `perfctr` kernel extension [12], which provides per-process access to the processor-specific performance monitoring counters. This allows us to accurately measure runtime performance based on the number of clock cycles, obtain information about branch misprediction rates, compute CPI, and so on.

In figures and tables, all reported code sizes are in bytes. Runtime performance, whenever not explicitly shown in clock cycles, has been normalized so that in charts the lower the bar, the better the performance. The characteristics of the nine benchmark programs are as follows: three of them (**barnes2**, **float_bm**, and **pseudoknot**) are floating-point intensive, one of them (**descrypt**) manipulates mostly bytes as it implements the DES encryption/decryption algorithm, and the remaining five manipulate integers, strings, and structured terms such as lists and trees. One benchmark, **md5**, creates large numbers of 32-bit integers. When tagged, they do not fit in 32-bit machine words, so on x86 they are boxed and stored on the heap as "bignums". They do fit in 64-bit machine words however, providing a significant advantage for AMD64.

### 6.1 Code Size Increase

Object code size typically increases on AMD64 compared with x86, but decreases are also possible due to e.g., the availability of more registers which results in less code for handling spilled temporaries. In HiPE/AMD64, the bulk of the code increase is due to the REX prefixes needed to generate 64-bit instructions and access the high registers. The rest of the increase is due to larger immediate offsets (for example, stack frames are often larger, requiring 32-bit offsets instead of 8-bit offsets when accessing data on the stack), and accessing

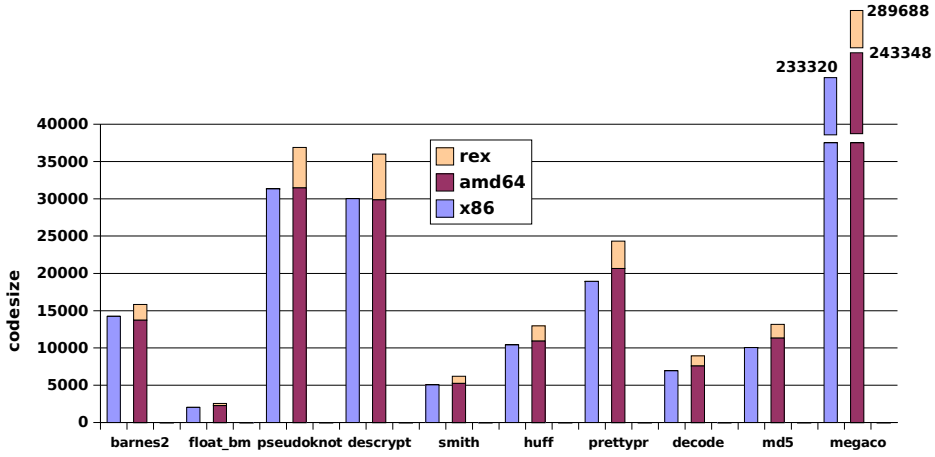



**Fig. 1.** Code size on x86 vs. AMD64 (size due to REX prefixes explicitly shown).

64-bit constants (which require a move to a register before each use). As can be seen in Fig. 1, the code size increase is moderate and mostly due to REX prefixes. Percentage-wise, the size of REX prefixes is between 12–17% of the total AMD64 code for all benchmarks.[12] Since these benchmark programs are quite small, we also show the code size for a larger program, **megaco** (Media Gateway Controller). It also confirms these numbers.

### 6.2 Running 32-bit vs. 64-bit Applications

With this experiment we try to determine whether it is worth developing a native code compiler for AMD64 in the first place. A main advantage of AMD64 machines is that they can run 64-bit operating systems and 32-bit x86 machine code in *compatibility* mode. So, if one is not interested in having a 64-bit address space, why not simply run the code in this mode? Even though there are drawbacks (for example, only 8 registers are available), even in compatibility mode, an AMD64 runs at full speed (i.e., no emulation is involved).

Since the answer to this question very much depends on the sophistication of the compiler, we offer two views. Figure 2 shows performance of AMD64 vs. x86 code when using two different register allocators and keeping every other backend component the same. It is clear that the 64-bit mode is a winner. It behaves better when the allocator cannot prevent spilling on x86 (such is the case when using linear scan). It also provides better performance in programs which manipulate bytes (**descrypt**) and large integers (**md5**) as it avoids the restrictions of the x86. On the other hand, there are programs (e.g., **smith** and **prettypr**) where the performance is slightly worse on 64-bit mode due to pointer-based data structures such as lists and records becoming larger, and thus placing additional burdens on the memory and cache subsystems.

### 6.3 Choice of Register Allocator

The HiPE compiler is one of the few native code compilers with a choice of three global register allocators: one based on *iterated register coalescing* [6], a *Briggs-style graph col-*

[12] For comparison, Appendix A.1 shows code size increase in programs generated using `gcc`.

| Benchmark | 64-bit mode | 32-bit mode | Ratio |
|---|---|---|---|
| **descrypt** | 394450329 | 417732104 | 0.94 |
| **smith** | 1124120607 | 975115035 | 1.15 |
| **huff** | 2824795743 | 2817767486 | 1.00 |
| **prettypr** | 982338705 | 838201998 | 1.17 |
| **decode** | 2147561421 | 2325866898 | 0.92 |
| **md5** | 231344119 | 2062041044 | 0.11 |

(a) Using iterated register coalescing

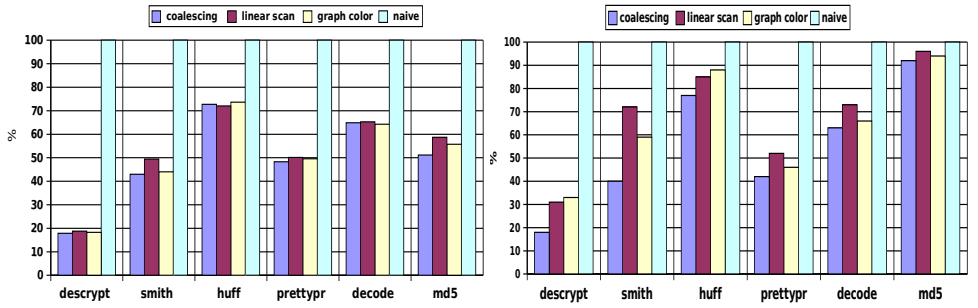
| Benchmark | 64-bit mode | 32-bit mode | Ratio |
|---|---|---|---|
| **descrypt** | 414822324 | 739868823 | 0.56 |
| **smith** | 1292385208 | 1744318644 | 0.74 |
| **huff** | 2797567542 | 3118378476 | 0.90 |
| **prettypr** | 1019246428 | 1039895068 | 0.98 |
| **decode** | 2160353131 | 2694052200 | 0.80 |
| **md5** | 265340924 | 2150536420 | 0.12 |

(b) Using linear scan register allocator

**Fig. 2.** Performance (clock cycles) of native 64-bit vs. 32-bit applications (i.e., x86 code).

*oring* allocator [3], and a *linear scan* register allocator [14, 15]. There is also a naïve allocator which keep temporaries in memory and only loads them into registers locally on a per-instruction basis; it however avoids loading temporaries when instructions can accept explicit memory operands. All four allocators were ported to AMD64.

Figure 3 shows normalized (w.r.t. the naïve allocator) performance results when varying the choice of allocator on AMD64 and x86. As can be seen, on both architectures, global register allocation really pays off; see e.g. **descrypt**.[13] On the other hand, since AMD64 has twice as many registers as x86, even a low-complexity algorithm such a linear scan provides decent performance and is competitive with graph coloring algorithms, which require significantly longer time to perform the allocation.




(a) On AMD64 (b) On x86

**Fig. 3.** Normalized performance of varying the register allocation algorithm on AMD64 and x86.

### 6.4 Reserving Registers for Parameter Passing

As can be seen in Fig. 4, choosing the right number of registers for parameter passing non-trivial. With the exception of **md5** whose performance improves by about 15%, the differences in performance (which is shown normalized w.r.t. using zero registers for parameter passing) are rather small. Since, as mentioned in Sect. 4.4, there may be other considerations (e.g. calling foreign code) when choosing the number of registers for parameter passing, we recommend taking these into account and choosing a number between 3 and 5.

[13] As can be seen in Fig. 6 (Appendix A.2), it also reduces the size of the generated native code.

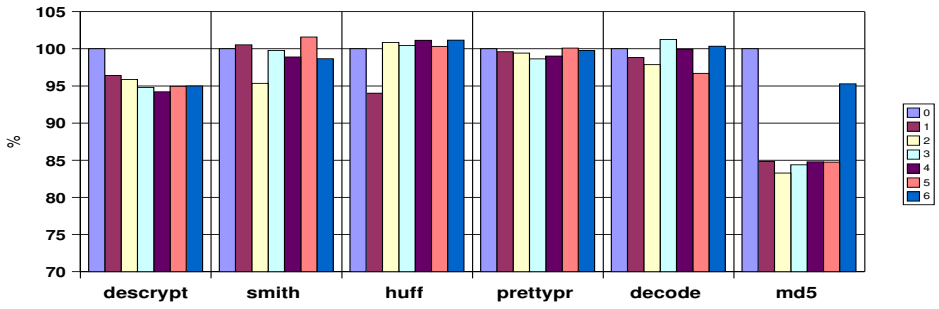



**Fig. 4.** Normalized performance varying the number of reserved registers for parameter passing.

### 6.5 Floating-Point Arithmetic

As can be seen in Table 1, there is a definite advantage to using SSE2 instead of `x87` for floating-point on AMD64. Erlang is not ideal for numerical applications: FP values are heap-allocated and never passed in registers, and FP register temporaries are short-lived, but even so there is still a moderate speedup by using SSE2.

**Table 1.** Performance comparison of SSE2 vs. x87 stack on AMD64.

| | Clock cycles | | | Code size | |
|---|---|---|---|---|---|
| Benchmark | using SSE2 | using x87 | SSE2/x87 | SSE2 | x87 |
| **barnes2** | 672921992 | 679159760 | 0.99 | 15828 | 16928 |
| **float_bm** | 601989479 | 792802467 | 0.76 | 2564 | 2468 |
| **pseudoknot** | 192459474 | 200751883 | 0.96 | 36880 | 38056 |

### 6.6 Use of Native Stack Pointer or Not

We saved the best for last. Figure 5 shows the branch misprediction rates on AMD64 and x86 when using the hardware-supported native stack vs. simulating the stack with a general-purpose register. Again, the message is clear: use of the native stack reduces the number of branch mispredictions[14] and executed clock cycles (data shown in Appendix A.3). Performance-wise, it pays off.

## 7 Related Work and Systems

The GNU Compiler Collection has mature support for AMD64 [7]. For AMD64 it includes optimizations such as: using direct `move` instructions instead of `push` or `pop` when changing the stack, preferring SSE2 over `x87`, using only the low 64 bits of SSE2 registers when possible, defaulting to a small code model, and replacing small 8 or 16-bit loads with 32-bit loads and explicit zero extensions. Instruction scheduling is implemented but found to be valuable mostly to SSE2 code. For calling conventions, `gcc` is bound to follow the ABI [8]. Both Intel and The Portland Group have released commercial C/C++/Fortran compilers with AMD64

[14] The lower branch misprediction rate for **md5** on AMD64 vs. x86 when using native stack is due to AMD64 not having to handle bignums by calling C routines.

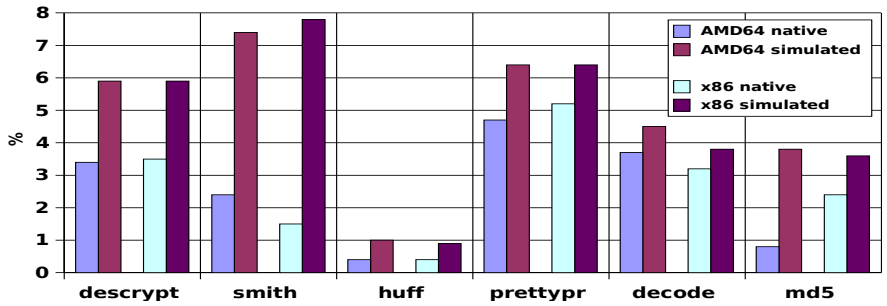



**Fig. 5.** Branch misprediction rates when using native vs. simulated stack.

support, but no detailed documentation about their implementation strategies appears to be available.

In the area of functional or declarative languages, very few directly support native code on AMD64 (we do not consider those that compile via C). The only one we know of to have mature AMD64 support, apart from HiPE, is O'Caml [10]. On AMD64, O'Caml passes 10 arguments in registers, uses the native stack, and reserves two registers for global variables: the heap pointer and the current exception handler. Its x86 backend passes 6 arguments in registers, uses the native stack, and reserves no registers for global variables. A straightforward graph coloring register allocator is used for both backends. The Glasgow Haskell Compiler has a preliminary AMD64 backend, but it currently does not implement any register allocation for AMD64.

## 8 Concluding Remarks

Although not as challenging as the IA64, AMD64 is a new 64-bit platform that offers a unique opportunity to compiler writers: the chance to "super size" their existing x86 backend with moderate effort. In this paper, we have described in detail how one can migrate an existing x86 backend to AMD64 and the issues that need to be addressed in order to obtain an efficient AMD64 backend. There are good indications that in the near future AMD64 machines might become as commonplace as x86 machines are today. If so, sooner-or-later, existing native code compilers will need to adapt to this architecture. We hold that our advice and measurements provide valuable guidance to those wishing to develop such a backend.

## Acknowledgments

This research and the development of the AMD64 backend of HiPE have 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.

## 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. 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.
4. 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.
5. R. K. Dybvig. *Three Implementation Models for Scheme*. PhD thesis, Department of Computer Science, University of North Carolina at Chapel Hill, 1987. Technical Report TR87-011. Available from: http://www.cs.indiana.edu/scheme-repository/.
6. L. George and A. W. Appel. Iterated register coalescing. *ACM Trans. Prog. Lang. Syst.*, 18(3):300–324, May 1996.
7. J. Hubička. Porting GCC to the AMD64 architecture. In *Proceedings of the GCC Developers Summit*, pages 79–105, May 2003.
8. J. Hubička, A. Jaeger, and M. Mitchell. *System V Application Binary Interface, AMD64 Architecture Processor Supplement*. See `www.x86-64.org`.
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*, 4(4):421–436, Aug. 2003.
10. X. Leroy et al. *The Objective Caml system release 3.07*. INRIA, Sept. 2003. See also http://caml.inria.fr/ocaml/.
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*, volume 2670 of *LNCS*, pages 134–149. Springer, Sept. 2002.
12. M. Pettersson. Linux x86 performance-monitoring counters driver. Available from: http://user.it.uu.se/~mikpe/linux/perfctr/.
13. 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*, volume 2441 of *LNCS*, pages 228–244, Berlin, Germany, Sept. 2002. Springer.
14. M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Trans. Prog. Lang. Syst.*, 21(5):895–913, Sept. 1999.
15. K. Sagonas and E. Stenman. Experimental evaluation and improvements to linear scan register allocation. *Software – Practice and Experience*, 33(11):1003–1034, Sept. 2003.
16. G. L. Steele Jr. LAMBDA: The ultimate declarative. MIT AI Memo 379, Massachusetts Institute of Technology, Nov. 1976.
17. G. L. Steele Jr. Rabbit: a compiler for Scheme (a study in compiler optimization). MIT AI Memo 474, Massachusetts Institute of Technology, May 1978. Master's Thesis.
18. M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. Prog. Lang. Syst.*, 13(2):181–210, Apr. 1991.

# A Additional Measurements

To support some claims in Sect. 6, we include additional measurements. The paper is self-contained without them and they should not be considered part of the submission.

## A.1 Code Size Increase

Table 2 shows the size of object files for some familiar Linux programs on AMD64. (We have also included the **beam** executable which contains code for the abstract machine and runtime system of Erlang/OTP.) These data were collected using the `size`, and `objdump` commands. In the table, the "Code size" column shows the total size as reported by the `size` command on AMD64 and the "REX%" column the part attributed to REX prefixes. The remaining columns show increase of various sections compared with the corresponding object files on x86. For example, the "text" increase is computed as $(amd64\_text - x86\_text)/x86\_text$, and similar calculations occur for obtaining numbers for "data", "bss", and "total".[15]

**Table 2.** Size of C object files (generated by gcc 3.3.3) on AMD64.

| | AMD64 | | Increase compared with x86 | | | |
|---|---|---|---|---|---|---|
| Application | Code size | REX% | text | data | bss | total |
| **xterm** | 314591 | 5.8% | 13.1% | 38.6% | 4.1% | 15.0% |
| **beam** | 1659007 | 9.7% | 30.0% | 48.8% | 84.1% | 43.0% |
| **gdb** | 2857930 | 6.6% | 18.2% | 88.4% | 15.1% | 19.4% |
| **ddd** | 3260866 | 7.7% | 0.6% | 70.1% | 33.8% | 3.2% |
| **emacs** | 6622446 | 10.1% | 14.9% | 66.3% | 0.0% | 50.5% |

Things to note are that in code generated by `gcc`, the REX prefix percentage is slightly less than the one we report in Sect. 6.1, but on the other hand, the total increase in code size in object files is often much bigger than that between HiPE/x86 and HiPE/AMD64. (For convenience, the data used to generate Fig. 1 are also shown in table form in Table 3.)

**Table 3.** Size of generated native code on x86 vs. AMD64.

| | x86 | AMD64 | | |
|---|---|---|---|---|
| Benchmark | Code size | Code size | REX | REX% |
| **barnes2** | 14236 | 15828 | 2102 | 13.3% |
| **float_bm** | 2036 | 2564 | 295 | 11.5% |
| **pseudoknot** | 31316 | 36880 | 5413 | 14.7% |
| **descrypt** | 30008 | 35988 | 6137 | 17.1% |
| **smith** | 5056 | 6208 | 960 | 15.5% |
| **huff** | 10416 | 12964 | 2046 | 15.8% |
| **prettypr** | 18916 | 24320 | 3658 | 15.0% |
| **decode** | 6948 | 8940 | 1352 | 15.1% |
| **md5** | 10044 | 13176 | 1846 | 14.0% |
| **megaco** | 233320 | 289688 | 46340 | 16.0% |

## A.2 Choice of Register Allocator

The effect of the register allocation algorithm used on the size of the generated code on AMD64 is shown in Fig. 6.

[15] In Table 2, one can *not* directly compare the total size increase with the 'REX%', since the total increase is based on the x86 code sizes and the 'REX%' is based on the AMD64 ones.

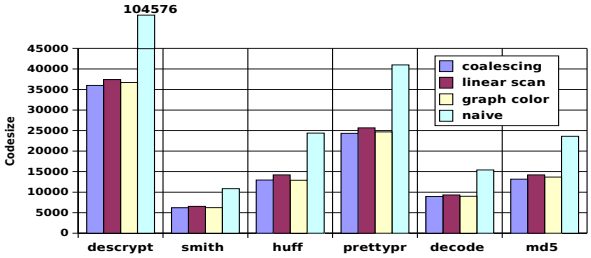



**Fig. 6.** Sizes of generated native code with different register allocators on AMD64.

### A.3 Use of Native Stack Pointer or Not

Data in Tables 4 and 5 show more detailed measurements than those of Sect. 6.6. They show the same branch misprediction rates as Fig. 5, but they also show runtime performance based on the number of clock cycles. With the exception of **huff** on AMD64 whose clock cycle increase we cannot fully explain (it is probably due to unlucky cache alignment), the numbers reinforce the message that using the processor's native stack rather than a simulated one is a winner.

**Table 4.** Performance using a native stack vs. a simulated stack on AMD64.

| | Branch misprediction % | | Clock cycles | | |
|---|---|---|---|---|---|
| Benchmark | native | simulated | native | simulated | ratio |
| **descrypt** | 3.4 | 5.9 | 394450329 | 427570517 | 0.92 |
| **smith** | 2.4 | 7.4 | 1124120607 | 1508869465 | 0.75 |
| **huff** | 0.4 | 1.0 | 2824795743 | 2614481627 | 1.08 |
| **prettypr** | 4.7 | 6.4 | 982338705 | 1041787437 | 0.94 |
| **decode** | 3.7 | 4.5 | 2147561421 | 2268147888 | 0.95 |
| **md5** | 0.8 | 3.8 | 231344119 | 267420561 | 0.87 |

**Table 5.** Performance using a native stack vs. a simulated stack on x86.

| | Branch misprediction % | | Clock cycles | | |
|---|---|---|---|---|---|
| Benchmark | native | simulated | native | simulated | ratio |
| **descrypt** | 3.5 | 5.9 | 417732104 | 451400978 | 0.93 |
| **smith** | 1.5 | 7.8 | 975115035 | 1331622536 | 0.73 |
| **huff** | 0.4 | 0.9 | 2817767486 | 2845832068 | 0.99 |
| **prettypr** | 5.2 | 6.4 | 838201998 | 882885782 | 0.95 |
| **decode** | 3.2 | 3.8 | 2325866898 | 2463384521 | 0.94 |
| **md5** | 2.4 | 3.6 | 2062041044 | 2197177458 | 0.94 |