Efficiently Compiling a Functional Language on AMD64: The HiPE Experience

Daniel Luna
luna@update.uu.se

Mikael Pettersson
mikpe@it.uu.se

Konstantinos Sagonas
kostis@it.uu.se

Department of Information Technology
Uppsala University, Sweden

ABSTRACT

We describe and document our experience from developing an AMD64 backend for the HiPE (High Performance Erlang) native code compiler. We consider implementation alternatives and critically examine design choices for obtaining an efficient AMD64 backend. In particular, we consider in detail how other functional language implementors can migrate their existing x86 backends to the AMD64 architecture, a platform which is becoming increasingly important these days. 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 information can save development effort for others who intend to engage in a similar feat.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, code generation, incremental compilers, optimization*; D.3.2 [**Programming Languages**]: Language Classifications—*Applicative (functional) languages*

General Terms

Experimentation, Languages, Measurement, Performance

Keywords

Functional programming, AMD64, Erlang

1. INTRODUCTION

It is hardly surprising that developing an efficient new backend for an existing native code 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 programming language implementors 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 developers. In this way, parameters that worked well in similar settings can be used "as is", and developers can concentrate their efforts in varying design choices that rely on those assumptions which are not valid in their framework.

In the context of the HiPE compiler, a native code compiler for the concurrent functional language Erlang,[1] we have spent a period of about a year developing and tuning a backend for AMD64. (Actually, this was the first 64-bit backend for 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 independent of the characteristics of the source language, we believe that our experiences and measurements are of interest to all declarative language implementors that consider developing an efficient AMD64 native code backend.

The contributions of this paper are:

- On the programming language implementation side, the paper considers the performance tradeoffs of various design alternatives for backend components on AMD64. In particular, we focus our attention on how functional languages with an existing x86 backend can migrate it 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 the choice of our design decisions.

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

[1]The Erlang/OTP (Open Telecom Plaform) system includes the HiPE compiler as an integated component. It is available both as open source and commercially from Ericsson. See `www.erlang.se` and `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 Opteron64 based servers and workstations; `www.sun.com/amd/`.

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 HIGH PERFORMANCE ERLANG COMPILER INFRASTRUCTURE

In this section, to set the context of our work, 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 (often in the order of 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.

The HiPE compiler currently has backends for SPARC V8+, x86, AMD64, and PowerPC. (The HiPE/AMD64 backend is available since October 2004.) 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: conditional constant propagation (ConstProp) [17], unreachable and dead-code elimination (DCE), and copy propagation (CopyProp). Finally, a type propagator guided by static type inference 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 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 (ConstProp, DCE, and CopyProp) as well as partial redundancy elimination (PRE) and control-flow optimizations 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 implementation decisions do not depend on characteristics of Erlang such as being concurrent or dynamically typed. They are thus applicable to other native code compilers for functional languages.

3. AN OVERVIEW OF AMD64

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

Architecturally, these processors are 64-bit machines, with 16 64-bit integer registers, 16 floating-point registers, 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.[3]

3.1 Technical Summary

Here we summarize the technical aspects of AMD64 that are relevant for language implementors. 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 64-bit general-purpose registers and 16 floating-point registers. 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

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

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. (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.

32-bit immediate operands on AMD64 are zero-extended when used in 32-bit operations (the default), but they are 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. GENERATING EFFICIENT CODE ON AMD64: GENERIC CONSIDERATIONS

For generation of efficient code on AMD64, there are a few general but important rules to obey:

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.

A direct 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.[4] A *code model* is a set of constraints on the size

[4] This is a generic issue on machines which only have immediate

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:[5]

Small code model All compile- 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.

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.[6]

On HiPE/AMD64 we opted for 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.

For comparison, 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.

HiPE/AMD64 implements both models. 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].[7]

operands smaller than their virtual address space. The issue also affects 32-bit SPARC and PowerPC, but not x86.

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

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

[7] 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 more lightweight strategies for register allocation, such as linear scan [14, 15], to become feasible on AMD64; this is especially important when compilation time is an issue, such as in JITs and in interactive systems.

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. It is even more important for functional programming languages due to their call-intensive nature. 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

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

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 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 Detecting and Avoiding REX Prefixes

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. EFFICIENCY CONSIDERATIONS SPECIFIC TO FUNCTIONAL LANGUAGES

5.1 Tail Recursion and Branch Prediction

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

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

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 [16]. Another approach is to push the return address on the stack *before* pushing the remaining actual parameters [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 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 state 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 state 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 state 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.

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

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 a result of such an asynchronous call.

  On Unix/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.

[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.

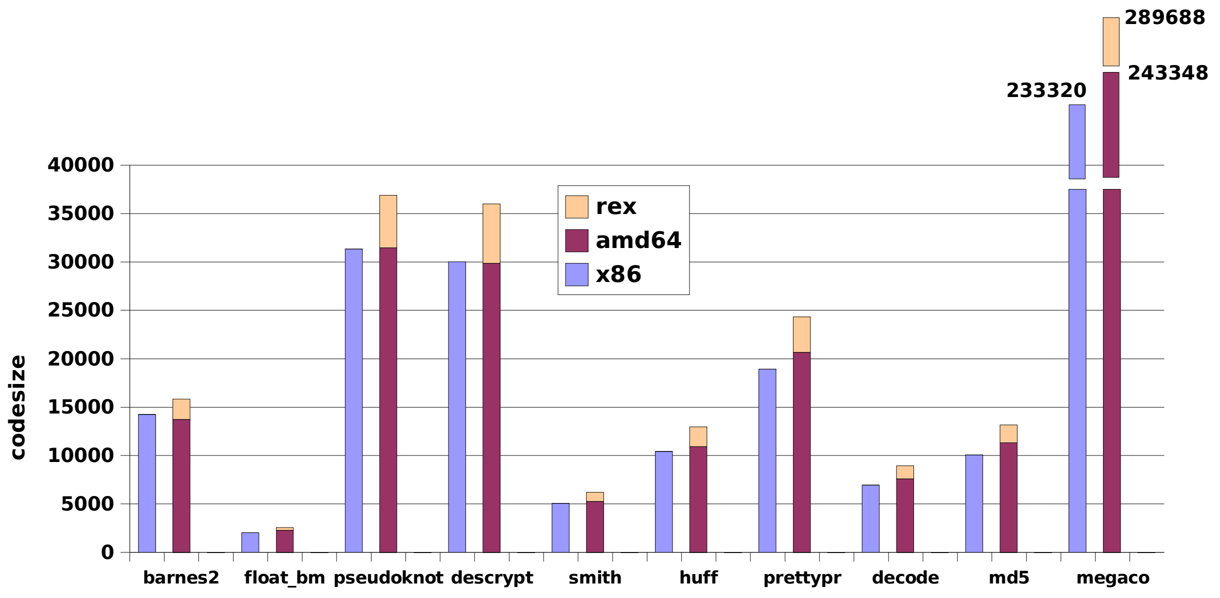


Figure 1: Code size in bytes on x86 vs. AMD64 (size due to REX prefixes explicitly shown).

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 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 64-bit constants (which require a move to a register before each use). As can be seen in Figure 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 "real-world" telecom library called **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 pro-

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

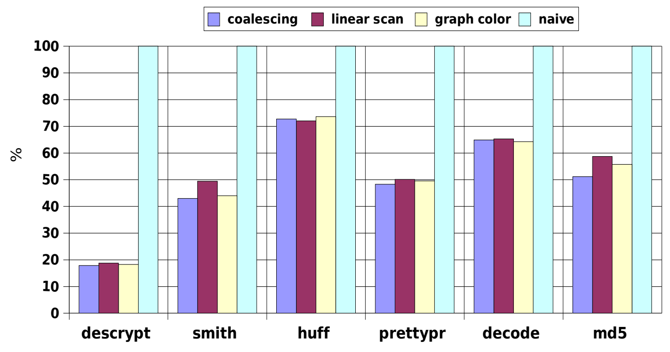
Benchmark	64-bit mode	32-bit mode	Ratio
descrypt	394450329	417732104	0.94
smith	1124120607	975115035	1.15
huff	2824795743	2817767486	1.00
prettypr	982338705	838201998	1.17
decode	2147561421	2325866898	0.92
md5	231344119	2062041044	0.11

(a) Using iterated register coalescing

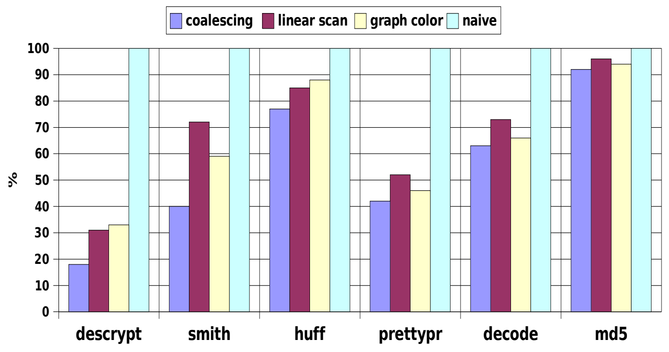
Benchmark	64-bit mode	32-bit mode	Ratio
descrypt	414822324	739868823	0.56
smith	1292385208	1744318644	0.74
huff	2797567542	3118378476	0.90
prettypr	1019246428	1039895068	0.98
decode	2160353131	2694052200	0.80
md5	265340924	2150536420	0.12

(b) Using a linear scan register allocator

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



(a) On AMD64



(b) On x86

Figure 3: Normalized performance of varying the register allocation algorithm on AMD64 and x86.

grams 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 coloring* 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.

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

6.4 Reserving Registers for Parameter Passing

As can be seen in Figure 4, choosing the right number of registers for parameter passing is 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.

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.

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]

[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.

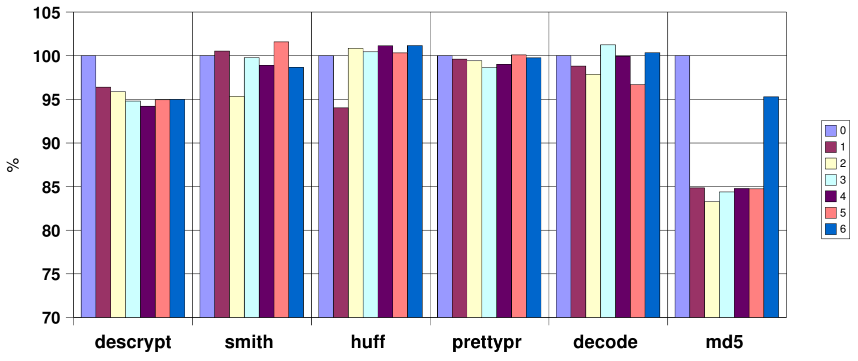


Figure 4: Normalized performance varying the number of reserved registers for parameter passing.

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

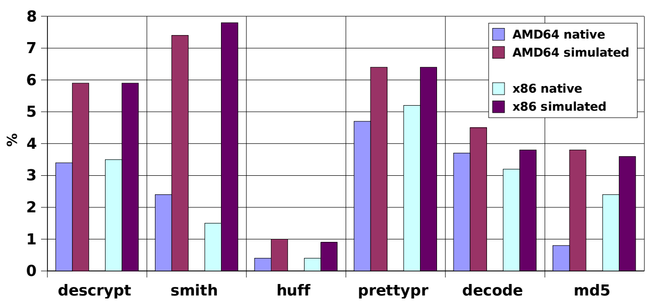


Figure 5: Branch misprediction rates when using native vs. simulated stack.

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

In the area of functional or declarative languages, currently 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 state 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 handling global state. 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. Plans for developing AMD64 backends for Clean and MLton are underway.

8. CONCLUDING REMARKS

Due to its similarities with the x86, the AMD64 is a new 64-bit platform that offers a unique opportunity to functional language implementors: 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 for (declarative) programming languages will need to adapt to this architecture. We hold that our experience and measurements, which correspond to a lot of work, 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.

9. 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 performance-monitoring counters kernel extension. 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. Rabbit: a compiler for Scheme (a study in compiler optimization). MIT AI Memo 474, Massachusetts Institute of Technology, May 1978. Master's Thesis.

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

Table 3: Size (in bytes) of generated native code on HiPE/x86 vs. HiPE/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%

APPENDIX

A. ADDITIONAL MEASUREMENTS

To support some claims in Sect. 6, we include additional measurements.

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]

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 Figure 1 are also shown in table form in Table 3.)

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 Figure 6.

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 Figure 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.

[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.

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%

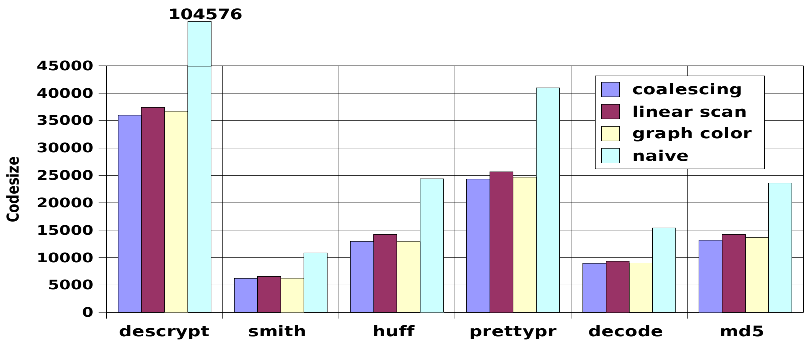


Figure 6: Sizes of generated native code with different register allocators on AMD64.

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