Porting GCC to the AMD64 architecture

Jan Hubička
SuSE ČR, s. r. o.
jh@suse.cz, http://www.ucw.cz/~hubicka

Abstract

In this paper we describe our experience from porting GCC to the AMD64 architecture and the AMD Opteron processor. Our target was a high quality port producing fast code. We discuss decisions taken while designing the Application Binary Interface (ABI) and effect of various code optimizations we implemented. We also present several open issues we would like to solve in the future.

1 AMD64 Instruction Set Overview

The AMD64 architecture [AMD64] is an extension of x86 instruction set to enable 64-bit computing while remaining compatible with existing x86 software. The CPU can operate in 64-bit mode, where semantic of several x86 instructions has been changed. Most notably:

- Single byte encoding of `inc` and `dec` instructions is no longer available. Instead the opcodes are used to encode a new prefix `REX` with four one-bit arguments. First argument is used to overwrite instruction operand size into 64 bits. Other three are used to increase amount of general purpose registers from 8 to 16.
- New 64-bit addressing mode is used by default. Prefix is available to overwrite into 32-bit addressing when needed.
- One of multiple possible encodings of direct addressing has been changed into instruction pointer relative addressing. Instruction pointer relative addressing is now one byte shorter than direct addressing.
- Default operand size remains 32-bit, however stack manipulation instructions, such as `push` and `pop` defaults to 64-bit operand size.
- The immediate operands of instructions has not been extended to 64 bits to keep instruction size smaller, instead they remain 32-bit sign extended. Additionally the `movabs` instruction to load arbitrary 64-bit constant into register and to load/store integer register from/to arbitrary constant 64-bit address is available.
- Several new instructions have been added to allow 64-bit conversions of data types.

Unlike earlier 64-bit architectures GCC has been ported to, some AMD64 features are unique, such as CISC instruction set, generally usable IP relative addressing, partial support for 64-bit immediate operands and more.

2 Application Binary Interface

Since GCC has been one of the first compilers ported to the platform, we had a chance to design the processor specific part of the application binary interface [AMD64-PSABI] from scratch. In this section we discuss the decisions we made and rationale behind them. We also discuss the GCC implementation, as well as problems we encountered while porting the software.

Majority of [AMD64-PSABI] has been designed in the early stages of development with just preliminary implementation of AMD64 support in GCC and no hardware nor simulator available. Thus we had just limited possibilities for experiments and most of our decisions has been verified by measuring of executable files sizes and number of instructions in them.

We never made serious study on how these relate to the performance, but it may be expected that the relation is pretty direct in the cases we were interested in.

2.1 Fundamental Types

We do use 64-bit pointers and `long`. The type `int` is 32-bit. This scheme is known as LP64 model and is used by all 64-bit UNIX ports we are aware of.

64-bit pointers bring expansion of the data-structures and increase memory consumption of the applications. A number of 64-bit UNIX ports also specify a code model with 32-bit pointers, LP32. Because of large maintenance cost of extra model (change of pointer size requires kernel emulation layer and brings further difficulties) and because of support for native 32-bit applications we decided to concentrate on LP64 support first and implement LP32 later only if necessary. See also Section 4.1 for some further discussion.

We considered the `long long` type to be 128-bit, since AMD64 has limited support for 128-bit arithmetics (that comes from extending support for 32-bit arithmetic in 16-bit 8086), however there are many programs that do expect `long long` to be exactly 64-bit, thus we specify optional `__int128` instead. At the moment no library functions to deal with the type are specified so it's usage in C environment may be uncomfortable. This is something we may consider to address in future extension of the ABI document.

The size of `long double` is 128 bits with only first 80 bits used to match native x87 format. The rest is just padding to keep long double values 128-bit aligned so loads and stores are effective. The padding is undefined that may bring problems when one is using `memcmp` to test for equality of two `long double` values.

Additionally we specify `__m64` and `__m128` types for SIMD operations.

All types do have natural alignment. ([i386-ABI] limits the alignment to 32-bit that brings serve performance problems when dealing with `double`, `long double`, `__m64` and `__m128` types on modern CPU.) It is allowed to access misaligned data of all types with the exception of `__m128`, since CPU traps on misaligned 128-bit memory accesses.

GCC Implementation

Our GCC implementation does support all specified types with the exception of `__float128`. At the moment GCC is not ready to support two extended floating point formats having the same size and thus implementing it would require considerable effort.

The 128-bit arithmetics patterns are also not implemented yet so code generated for `__int128` is suboptimal.

2.2 The Stack Frame

Figure 1: Stack Frame

Size	Contents	Frame
0–8n	incoming arguments	Previous
8	return address	Current
0,8	previous `%rbp` value	
0,8	padding	
?	local data	
?	register spill area	
0–4	padding	
0–48	register save area	
0,8	padding	
0,8	padding	Allocated via `push`
0–8n	outgoing arguments	

Unlike [i386-ABI] we do not enforce any specific organization of stack frames giving compiler maximal freedom to optimize function prologues and epilogues. In order to allow easy spilling of x87 and SSE registers we do specify 128-bit stack alignment at the function call boundary, thus function calls may need to be padded by one extra `push` since AMD64 instruction set naturally aligns stack to 64-bit boundary only.

We additionally specify the red zone of 128 bytes below the stack pointer function can use freely to save data without allocating the stack frame as long as the data are not required to survive function call.

The sample stack frame organization based on extending the usual IA-32 coding practice to 64-bit is shown at Figure 1, the sample prologue code is shown at Figure 2.

GCC Implementation

We found the use of frame pointer and `push`/`pop` instructions to be common bottleneck for the function call performance. The AMD Opteron CPU can execute

Figure 2: Function Prologue and Epilogue

```
push   %rbp            Save frame pointer
movq   %rsp %rbp       Initialize frame pointer
subq   $48, %rsp       Allocate stack frame
pushq  %rbx            Save non-volatile registers
pushq  %r12            clobbered by function
pushq  %r13
...                    Function body
popq   %r13            Restore registers
popq   %r12
popq   %rbx
leave                  Restore %rbp
                       and deallocate stack
ret
```

Figure 3: Stack Frame in GCC

Size	Contents	Frame
0–8n	incoming arguments	Previous
8	return address	
0,8	previous %rbp value	
0–48	register save area	
0,8	padding	Current
0–96	va-arg registers	
?	local data	
?	register spill area	
0–8	padding	
0–8n	outgoing arguments	

stores at the rate of two per cycle, while it requires 2 cycles to compute new `%rsp` value in `push` and `pop` operations so the sequence of `push` and `pop` operations executes 4 times slower.

We reorganized the stack frame layout to allow shorter dependency chains in the prologues and epilogues as shown on Figure 3. To save and restore registers we commonly use the sequence of `mov` instructions and we do allocate whole stack frame, including outgoing argument area, using single `sub` opcode as shown in Figure 4. AMD Opteron processor executes the prologue in 2 cycles, while the usual prologue (Figure 2) requires 9 cycles. Similarly for the epilogues.

Unfortunately the produced code is considerably longer — the size of `push` instruction is 1 byte (2 bytes for extended register), while the size of `mov` is at least 5 bytes. In order to reduce the expenses, GCC does use profile

Figure 4: GCC Generated Prologue and Epilogue

```
movq   %rbx,-24(%rsp)  Save registers
movq   %r12,-16(%rsp)
movq   %r13,-8(%rsp)
subq   $72, %rsp       Allocate stack frame
...                    Function body
movq   48(%rsp),%rbx   Restore registers
movq   56(%rsp),%r12
movq   64(%rsp),%r13
addq   $72, %rsp       Deallocate stack frame
ret
```

information to use short sequences in the cold function. Additionally it estimates number of instructions executed per one invocation of function and use slow prologues and epilogues when it exceeds given threshold (20 instructions for each saved register).

We found heuristics choosing between fast and short prologues difficult to tune — the prologue/epilogue size is most expensive for small functions where it also should be as fast as possible. As can be seen in the Table 7, the described behavior results in about 1% speedup at the 1.1% code size growth ("prologues using moves" benchmark). Bypassing the heuristics and using moves for all prologues results in additional speedup of 1% and additional 1.1% code size growth ("all prologues using moves" benchmark). The heuristics works better with profile feedback (Table 9). This is something we should revisit in the future.

GCC does always eliminate the frame pointer unless function contain dynamic stack allocation such as `alloca` call. This always result in one extra general purpose register available and fewer instructions executed.

Contrary to the instruction counts, eliminating of frame pointer may result in larger code, because `%rsp` relative addressing encoding is one byte longer than `%rbp` relative one. Thus it may be profitable to not eliminate frame pointer when function do contain many references to the stack frame. Command line option `-fno-omit-frame-pointer` can be used to force use of frame pointer in all functions.

For 64-bit code generation omitting frame pointer results in both smaller and faster code on the average (Tables 7, 8, 9 and 10). In the contrary, for 32-bit code generation it results in code size growth (Tables 11 and 12). This is

caused by the fact that increased register file and register argument passing conventions eliminated vast majority of stack frame accesses produced by the 32-bit compiler.

In GCC stack frame layout the register save area and local data are reordered to reduce number of instruction when `push` instructions are used to save registers — the stack frame and outgoing arguments area allocation/deallocation can be done at once using single `sub`/`add` instruction. The disadvantage is that `leave` can not be used to deallocate stack frame in combination with `push` and `pop` instructions. In our benchmarks the new approach brought noticeable speedups for 32-bit code, however it is difficult to repeat the benchmarks since the prologue/epilogue code is dependent on the new stack frame organization and would require some deeper changes to work in the original scheme again.

At the moment GCC is just partly taking advantage of the red zone. We do use red zone for leaf functions having data small enough to fit in it and for saving some temporarily allocated data in instruction generation (so the `sub` and `add` instructions in Figure 4 would be eliminated for leaf functions). For the benefit of kernel programming (signal handlers must take into account the red zone increasing stack size requirements), option `-fno-red-zone` is available to disable usage of red zone entirely.

As can be seen in the Tables 7 and 8, red zone results in slight code size decrease and speedups. The effect depends on how many leaf functions require stack frame. This is uncommon for C programs, but it happens more frequently in template heavy C++ code where function bodies are large due to in-lining (Tables 10 and 9).

We do not use the red zone for spilling registers nor for storing local variables in non-leaf functions as GCC is not able to distinguish between data surviving function calls and data that does not. Extending GCC to support it may be interesting project and may reduce stack usage of programs, however we have no data on how effective the change can be.

To further reduce the expenses, GCC does schedule the prologue and epilogue sequence to overlap with function body. In the future we also plan to implement shrink-wrapping optimization as the expense of saving up to 6 registers may be considerable.

2.3 Stack Unwinding Algorithm

To allow stack unwinding, we do use additional information saved in the same format as specified by DWARF debugging information format [DWARF2]. Instead of `.debug_frame` section specified by DWARF we do use `.eh_frame` section so the data are not stripped.

The DWARF debugging format defines unwinding using the interpreted stack machine describing algorithms to restore individual registers and stack frames. This mechanism is very generic and allows compiler to do pretty much any optimization on stack layout it is interested in. In particular we may eliminate stack frame pointer and schedule prologues and epilogues into the function body.

The disadvantage is the size of produced information and speed of stack unwinding.

GCC Implementation

Implementation in GCC was straightforward as DWARF unwinding was already used for exception handling on all targets except for IA-64. We extended it by support for emitting unwind info accurate at each instruction boundary (by default GCC optimize the unwind table in a way so it is accurate only in the places where exceptions may occur). This behavior is controlled via `-fasynchronous-unwind-tables`.

GCC perform several optimizations on the unwind table size and the tables are additionally shortened by assembler, but still the unwind table accounts for important portion of image file size.

As can be seen in the Table 7 it consumes, at the average, 7.7% of the stripped program binaries size, so use of `-fno-asynchronous-unwind-tables` is recommended for program where unwinding will never be necessary.

The GCC unwind tables are carefully generated to avoid any runtime resolved relocations to be produced, so with the page demand loading tables are never load into memory when they are not used and consume the disc space only.

Main problem are the assembly language functions. At the present programmer is required to manually write DWARF byte-code for any function saving register or having nonempty stack frame in order to make unwind-

ing work. This is difficult and most of assembly language programmers are unfamiliar with DWARF. It appears to be necessary to extend the assembler to support describing of the unwind information using the pseudo-instructions similar to approach used by [IA-64-ABI].

2.4 Register Usage

The decision on split in between volatile (caller saved) and non-volatile (callee saved) register presented quite difficult problem. The AMD64 architecture have only 15 general purpose registers and 8 of them (so called extended registers) require `REX` prefix increasing instruction size. Additionally the registers `%rax`, `%rdx`, `%rcx`, `%rsi` and `%rdi` implicitly used by several IA-32 instructions. We decided to make all of these registers volatile to avoid need to save particular register only because it is required by the operation. This leaves us with only `%rbx`, `%rbp` and the extended registers available for non-volatile registers. Several tests has shown smallest code to be produced with 6 global registers (`%rbx`, `%rbp`, `%r12`–`%r15`).

Originally we intended to use 6 volatile SSE registers, however saving of the registers is difficult: the registers are 128-bit wide and usually only first 64-bits are used to hold value, so saving registers in the caller is more expensive.

We decided to delay the decision until hardware is available and run several benchmarks with different amount of global registers. We also experimented with the idea of saving only lower half of the registers. Our experiments always did lead to both longer and slower code, so in the final version of ABI all SSE registers are volatile.

Finally the x87 registers must be volatile because of their stack organization and the direction flag is defined to be clear.

2.5 Argument Passing Conventions

To pass argument and return values, the registers are used where possible. Registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` and `%r9` are used to pass integer arguments. In particular, register `%rax` is not used because it is often required as special purpose register by IA-32 instructions so it is inappropriate to hold function arguments that are often required to be kept in the register for a long time. Registers `%xmm0`–`%xmm5` are used to pass floating point arguments. x87 registers are never used to pass argument to avoid need to save them in variadic functions.

To return values registers `%rax`, `%rdx`, `%xmm0`, `%xmm1`, `%st0` and `%st1` are used. The usage of `%rax` for return value seems to be considerable win even at the expense of extra `mov` instruction needed for functions returning copy of the first argument and functions returning aggregates in memory via invisible reference.

The aggregates (structures and unions) smaller than 16 bytes are passed in registers. The decision on what register class (SSE, integer or x87) to use to pass/return the aggregate is rather complicated; we do pass each 64-bit part of structure in separate register, with the exception of `__m128` and `long double`.

The argument passing algorithm classifies each field of the structure or union recursively into one of the register classes and then merge the classes that belongs to the same 64-bit part. The merging is done in a way so integer class is preferred when both integer and SSE is used and structure is forced to be passed in memory when difficult to resolve conflicts appears. The aggregate passing specification is probably the most complex part of the ABI and we hope that the benefits will outweight the implementation difficulties. For GCC it requires roughly 250 lines of C code to implement.

Arguments requiring multiple registers are passed in registers only when there are enough available registers to pass argument as a whole in order to simply `va_arg` macro implementation.

Variable sized arguments (available in GCC only as GNU extension) are passed by reference and everything else (including aggregates) is passed by value.

GCC Implementation

It is difficult to obtain precise numbers, but it is clear that the register passing convention is one of the most important changes we made relative to [i386-ABI] improving both performance and code size. The amount of stack manipulation is also greatly reduced resulting in shorter debug information. On the other hand, the most complex part, passing of aggregates, has just minor effect on C code. We believe it will become more important in future for C++ code.

At the moment GCC does generate suboptimal code in

number of cases where aggregate is passed in the multiple registers — the aggregate is often offload to memory in order to load it into proper registers. Beside that GCC should implement all nuances of argument passing correctly.

For functions passing arguments in memory, the stack space is allocated in prologue; deallocated in epilogue and plain `mov` operations are used to store arguments. This is in contrast to common practice to use `push` operation for argument passing to reduce code size. Despite that experimental results shows both speedups and code size reductions of the AMD64 binaries when `mov` instructions are used (See `-maccumulate-outgoing-args` in the Table 7, 8, 9, and 10). This is in sharp contrast to IA-32 code generation experience (Tables 11 and 12).

There are multiple reasons for the image size to be reduced. Usage of `push` instructions increases unwind table sizes (about 3% of the binary size). Most of the functions has no stack arguments, however they still do require stack frame to be aligned. This makes GCC to emit number of unnecessary stack adjustments. Last reason seems to be fact that majority of values passed on the stack are large structures where GCC is not using push instructions at all.

2.6 Variable Argument Lists

More complex argument passing conventions require nontrivial implementation variable argument lists. The `va_list` is defined as follows:

```
typedef struct {
   unsigned int gp_offset;
   unsigned int fp_offset;
   void *overflow_arg_area;
   void *reg_save_area;
} va_list[1];
```

The `overflow_arg_area` points to the end of incoming arguments area. Field `reg_save_area` points to the start of register save area.

Prologue of function then uses 6 integer moves and 6 SSE moves to save argument registers. In order to avoid lazy initialization of SSE unit in the integer only programs, hidden argument in the register `%al` is passed to functions that may use variable argument lists specifying amount of SSE registers actually used to pass arguments.

We decided to use the array containing structure for `va_list` type same way as [PPC-ABI] do to reduce expenses of passing `va_list` to the functions — arrays are passed by reference, while structures by value. This is valid according to the C standard, but brings unexpected behavior in the following function:

```
#include <stdarg.h>
void t (va_list *);
void q (va_list a)
{
  t(&a);
}
```

The function `t` expects address of the first element in the array, while in the second one, the array argument is merely an shortcut for a pointer so it passes pointer to the pointer to the first argument. This unexpected behavior did not trigger in Open Source programs since these already has been cleaned up to work on Power-PC, but has been hit by proprietary software vendors who claimed this to be a compiler bug even when GCC correctly emit an warning message “passing arg 1 of ‘t’ from incompatible pointer type”

GCC Implementation

The register save area is placed on fixed place in stack frame as shown in Figure 3. There is no particular reason for that, but it was slightly easier to implement in GCC.

The computed jump is used in the prologue to save only registers needed. This results in small savings for programs calling variadic function with floating point operands, but makes program calling variadic functions using non-variadic prototypes to crash. Such programs are not standard conforming, but they happen in practice. We noticed the problem for strace and Objective C runtime. We may consider replacing the jump table by single conditional to avoid such crashes.

Second important compatibility problems arrises from implicit type promoting. All 64-bit targets supported by SuSE Linux do promote operands to 64-bit values and several packages depend on it. Most notable example is GNOME. While promoting all function operands to 64-bit would be too expensive, we may consider promoting the operands of variadic functions to avoid such compatibility issues.

2.7 Code Models

The 32-bit sign extended immediates and zero extending loads of the immediate allows convenient addressing of only first 2^{31} bytes of the address space. The other areas needs to be addressed via `movabs` instructions or instruction pointer relative addressing. In order to allow efficient code generation for programs that do fit in this limitation (almost all programs today) we define several code models:

small All relocations (code and data) are expected to fit in the first 2^{31} bytes. This is the default model GCC use. This code model can be produced via `-mcmodel=small` command line option.

kernel All relocations (code and data) are expected to fit in the last 2^{31} bytes. This is useful for kernel address space to not overlap with the user address space. This code model can be produced via `-mcmodel=kernel` command line option.

medium Code relocations fit in the first 2^{31} bytes and data relocations are arbitrary. This code model can be produced via `-mcmodel=medium` command line option. The medium code model has significant code size (about 10%) and noticeable performance (about 2%) penalty (see Tables 7, 8, 9 and 10). These penalties are larger than the authors expectations and probably further improvements to the GCC code generation are possible.

large Code relocations and data relocations are arbitrary. This model is currently not supported by GCC as it would require to replace all direct jumps via indirect jumps. We don't expect this model to be needed in foreseeable future. Large programs can be split into multiple shared libraries.

The position independent code generation can be effectively implemented using the instruction pointer relative addressing. We implemented scheme almost identical to IA-32 position independent code generation practices only replacing the relocations to option global offset table address and index in it by single instruction pointer relative relocation. Similarly the instruction pointer relative addressing is used to access static data.

The resulting code relies on the overall size of the binary to be smaller than 2^{31} bytes. An [AMD64-PSABI] extension will be needed in the case this limitation will become a problem. The performance penalty of `-fpic` is about 6% on AMD64 compared to 20% on IA-32 (see Tables 9, 10, 11 and 12).

3 Implemented Optimizations

In this section we describe target specific optimizations implemented for the first hardware implementation of AMD64 architecture — the AMD Opteron CPU.

The AMD Opteron CPU core has rather complicated structure. The AMD64 instructions are first decoded and translated into micro operations and passed to separate integer and floating point on chip schedulers. Integer instructions are executed in 3 symmetric pipes of overall depth 11 with usual latency of 1 cycle, while floating point instructions are issued into 3 asymmetric pipes (first executing floating point add and similar operations, second having support for long latency instructions and multiple and third executing loads and stores). For more detailed description see also [Opteron].

The processor is designed to perform well on the code compiled for earlier IA-32 implementation and thus has reduced dependency on CPU model specific optimizations. Still several code generation decisions can be optimized as described in detail in [Opteron]. We implemented majority of these and here we describe only those we found most effective.

As can be seen in the Table 11, enabling AMD Opteron tuning via `-march=k8` improves integer program performance by about 10% relative to compiler optimizing for i386. Relative to the compiler optimizing for Pentium-Pro the speedup is only about 1.1%. The optimizations common for Pentium-Pro and Opteron include the scheduling (scheduling for Pentium-Pro still improves Opteron performance), avoiding of memory mismatch stalls, use of new conditional move and `fcomi` instructions and `-maccumulate-outgoing-args`.

For floating point programs the most important optimization is use of SSE instruction set (10%) followed by the instruction scheduling (not visible in the Table 12 because the x87 stack register file does not allow effective scheduling, but noticeable in the Table 10).

3.1 Integer Code Instruction Selection

Majority of IA-32 instructions generated by today compilers are well implemented in the Opteron core so the code generation is straightforward.

In the Tables 7, 8, 9 and 10, "full sized loads and moves"

refers to the transformation of 8-bit and 16-bit loads into zero extensions; use of 32-bit reg-reg moves for moving 8-bit and 16-bit values and symmetric change for SSE. The transformation is targeted to avoid hidden dependencies in the on-chip scheduler. The transformation has important effect for SSE code and smaller but measurable effect on code manipulating with 8-bit and 16-bit values.

Second important optimization we implemented is elimination of use `push` and `pop` instructions as mentioned in Section 2.2 and 2.5

Other optimization implemented had just minor effect on overall performance.

3.2 SSE floating point arithmetics

Unlike integer unit, the floating point unit has longer latencies (majority of simple floating point operations takes 3 cycles to execute) and is more sensitive to instruction choice.

The operations on whole SSE registers are usually more expensive than operations on the 64-bit halves. This holds for the move operations also, so it is desirable to always use partial moves when just part of SSE register is occupied (this is common for scalar floating point code). In particular it is desirable to use `movlpd` instead of `movsd` to load double precision values, since `movsd` does clear upper half of the register. `movsd` is the used for register to register moves. This remains the upper half of register undefined that may cause problem when the register is used as a whole for instance for logical operation that has no scalar equivalent. The CPU internally keeps values in different format depending on how they are produced (either single, double precision or integer) when register is in wrong format, serve reformatting penalty occurs.

In order to eliminate reformatting penalties we do reformat the register explicitly before each such operation (fortunately the logical operations are rare in generated code as they are used for conditional moves and fabs/neg expansion only) using `movhlpd`. In the future it may be interesting to implement special pass inserting the conversions only when they are actually needed as most of the `movhlpd` instructions emit are redundant. See "partial SSE register moves" in the Tables 7, 8, 9 and 10) for the comparison of this code generation strategy to the usual one recommended by [Pentium4].

For single precision scalars the situation is different. There is no way conveniently to load single precision data into memory without clearing the upper part of register (`movlps` require 64-bit alignment) and thus we maintain the whole registers in single precision. In particular we do use `movss` to load values and `movaps` for register to register moves.

This scheme brings difficulties with `cvtsi2ss` and similar instructions that do rewrite the lower part only. In this case `xorps` is used first to clear the register. Again the large portion of `xorps` instructions issued this way are redundant because the register is already in specified format. The CPU also special case `cvtsd2ss` instruction where the bytes 4–8 of the register are reformatted to single precision too, however bytes 8–16 remains in the previous format. We risk the reformatting penalty here, since bytes 8–16 are rarely in the double precision format because of the use of partial moves described above. We plan to add an command line option to force issuing of the reformatting here. Also we may reconsider this decision in the case we implement the pass for smart placement of reformatting instructions. See Tables 7, 8, 9 and 10, and benchmark "full sized loads and moves" described in Section 3.1.

3.3 Scheduling

Implementation of instruction scheduling was difficult for several reasons. The AMD Opteron CPU has complicated pipeline expanding each operation into multiple micro operations renaming the register operands and executing them separately in rescheduled order. The available documentation is incomplete and the effect of instruction scheduling on such architectures does not appear to be well studied.

As can be seen in Table 10, instruction scheduling enabled via `-fschedule-insns2` remains one of the most important optimizations we implemented for floating point intensive benchmarks. On the other hand the effect is about 10 times lower than on the in-order Alpha CPU (Table 14).

GCC at the present implements only local basic block scheduling that is almost entirely redundant with the out-of-order abilities of the CPU. We experimentally implemented an limited form of trace scheduling and measured an improvement of additional 1% for the SPECfp. Our expectation is that the more global the GCC scheduler algorithm will be, the less redundancies with out-of-order core will be apparent, so the benefits of global

algorithms should be comparable to ones measurable on in-order CPUs.

Our implementation represents just a simplified model of the real architecture. We model the allocations of decoders, floating point unit (fadd, fmul and fstore), the multiplier and load store unit. We omit model of the reorder buffers — the micro instructions are assumed to be issued to the execution pipes immediately in the fixed model. This also allows us to omit model of the integer and address generation units as never more than 3 instructions are issued at once.

Most of the stalls the scheduler can avoid are related to loads and stores. In order to avoid the stall it is necessary to model the instruction latencies and the fact that address operands are needed earlier than the data operands. The scheduler can reorder the computations so the data operands are computed in parallel with loads. GCC scheduler does assume that all the results must be available in order to instruction be issued and thus we reduce the latencies of instructions computing values used as data operands of load-execute instructions by up to 3 cycles (the latency of address generation unit). Even when latencies of majority instructions are shorter than 3 cycles and thus we can not reduce the latency enough to compensate the load unit latency, this model is exact for the in-order simplification of CPU described above as the instruction computing data operands must be output before load-execute instruction itself.

4 Experimental Results

We present benchmarks of majority optimizations discussed. We also present the same benchmarks performed on IA-32 and Alpha system where possible to give an comparison of effectivity of individual optimizations on these architectures. We hope this to be useful to apply earlier published results on compiler optimization (such as [FDO]) to the new platform and give a guide of what optimizations are most important. We also present results with two different optimization levels — the standard optimization (`-O2`) used by the majority of distributions today and aggressive optimization (`-O3 -ftracer -funroll-loops -funit-at-a-time` with profile feedback) we found to give best overall SPEC score.

We did use modified prerelease of GCC 3.3 as used by SuSE Linux 8.2 for AMD64. All the runs were performed on SuSE Linux on dedicated machines, however important amount of random noise remains (especially for benchmarks Mesa, Gzip, Perl and Twolf). Due to time limitations the benchmarks were performed with one iteration only except for the benchmarks in the Table 9 and 10 that were computed with 3 iterations. Because the runs were not done on final hardware and because we didn't satisfy the conditions for reportable runs in all tests, we present relative numbers only.

Each table is divided into two sections — first part includes optimizations enabled by default at given optimization level, while the other part contains optimization that user needs to enable by hand either because they are ineffective, inappropriate for given settings or does not obey the language standards. Each table also contains comparison of two runs with equal settings in the first line to present rough approximation of the noise in the numbers. Both performance and sizes of the stripped binaries are presented. The numbers always represent relative speedup (or code size increase) from the run with the specified feature disabled to the run with specified feature enabled. For instance `-fomit-frame-pointer` run in the table 7 compare performance of `-O2 -fno-omit-frame-pointer` to `-O2 -fomit-frame-pointer`. The benchmark "standard optimization" compare `-O0` to `-O2`.

The Following benchmarks were performed:

aggressive optimization compare performance of unoptimized code (`-O0`) to the aggressive optimization settings described above.

all prologue using move eliminate use of all `push` and `pop` operations in the prologues and epilogues except for cases where single register is saved. See Section 2.2.

-fasynchronous-unwind-tables enable production of DWARF2 unwind information. See Section 2.3.

`-fbranch-probabilities` enable profile feedback based optimizations. We implemented majority of transformations described on [FDO] with the exception of function in-lining and switch statement expansion.

`-fgcse` enable global optimizers including (limited form of) partial redundancy elimination, load motion, constant propagation and copy propagation. GCC does contain loop invariant hoisting and extended basic block based value numbering pass making the global optimizers partly redundant.

`-fguess-branch-probability` enable optimizations driven by static profile estimation. The profile is estimated by methods based on [profile] when profile feedback is not available.

`-finline-functions` enable function in-lining.

`-fold-unroll-loops` enable old loop unroller that actually unrolls some loops on Alpha.

`-fomit-frame-pointer` enable elimination of frame pointer by using stack pointer instead. See Section 2.2.

`-foptimize-sibling-calls` transform call to leaf function into jump.

`-fpeel-loops` enable loop peeling.

`-fpic` produce position independent code. See Section 2.7.

`-freorder-blocks` enable intra-function basic block reordering and duplication based on significantly modified software trace cache algorithm [STC].

`-fschedule-insns2` enable post-register allocation local scheduling. See Section 3.3.

`-fschedule-insns` enable pre-register allocation region scheduling (not available for IA-32 and AMD64).

`-fstrength-reduce` enable strength reduction.

`-fstrict-aliasing` enable ANSI-C type based aliasing.

full sized loads and moves avoids use of instructions initializing just portion of the destination registers. See Section 3.2 and 3.1.

`-ftracer` enable super-block formation using algorithm similar to [FDO]. The super-blocks are unified again after optimizations by cross-jumping pass so this transformation is not used to improve scheduling as commonly described in the literature. It is aimed to improve CSE and other transformation by simplifying the control flow.

`-funit-at-a-time` enable optimizations on whole compilation unit. At the moment GCC perform stronger function in-lining (in-lining of small functions called before defined and static functions called once) and use register calling conventions for static functions on IA-32. Only effective for C compiler.

Table 1: Compilation Time Cost (AMD Opteron)

options	slowdown
	0.00%
`-fstrict-aliasing`	−1.13%
`-fasynchronous-unwind-tables`	−0.38%
`-freorder-blocks`	0.00%
`-fomit-frame-pointer`	0.37%
`-mred-zone`	0.38%
`-mfpmath=sse`	0.75%
`-maccumulate-outgoing-args`	0.75%
`-foptimize-sibling-calls`	0.76%
`-fguess-branch-probabilities`	1.54%
`-fschedule-insns2`	2.33%
`-fgcse`	6.88%
`-ffast-math`	−1.88%
`-ftracer`	0.00%
`-frename-registers`	0.74%
`-funroll-loops`	3.38%
`-fpic`	3.39%
`-funroll-all-loops`	5.32%
`-mcmodel=medium`	2.27%
`-fbranch-probabilities`	142.74%

`-funroll-all-loops` enable loop unrolling of all small enough loops in the hot spots.

`-funroll-loops` enable loop unrolling for loops with known induction variable. While working on the paper we noticed that our new implementation has important flaw avoiding loops from being unrolled on Alpha architecture.

`-m64` enable 64-bit code generation (used in comparisons relative to IA-32 code).

`-mfpmath=sse` eliminate use SSE(2) instruction set for scalar floating point calculations.

`-mcmodel` controls code and data segment size limits. See Section 2.7.

`-mred-zone` enable use of 128 bytes below stack pointer for local data. See Section 2.2.

partial SSE moves eliminate use of `movlpd` for double precision loads and `movsd` for register to register moves. See Section 3.2.

prologue using move eliminate use of hot `push` and `pop` operations in the prologues and epilogues. See Section 2.2.

standard optimization compare performance of unoptimized code (`-O0`) to the standard optimization settings (`-O2`).

Table 2: Desktop Performance Relative to 32-bit System

test	speedup
bootup time	−0.9%
KDE startup from disk	18.1%
KDE startup from cache	14.6%

Table 3: Gimp Compilation Times Relative to 32-bit System

	speedup		
test	real	user	system
`tar xjf`	17.7%	9.8%	4%
`./configure`	−4.3%	0.7%	−31%
`make`	12.9%	19.8%	−39%

4.1 Real World Performance

One of the main goals has been to develop system ready for both enterprise and desktop (workstation) use. While the need of 64-bit addressing space for the enterprise is well understood, the effect on desktop performance is often discussed. The main drawback of 64-bit system, as discussed in section 2.1 is the increased memory footprint of the programs and subsequent slowdown of program startup times critical for today desktop systems.

In this section we present few simple benchmarks of this phenomenon on SuSE Linux 8.2. Both the 32-bit and 64-bit version of the system were installed on the equally sized ReiserFS partitions in the default configuration. The tests were performed in the same order on both systems with reboots in between. Additional packages were installed as needed. We hope this procedure to minimize amount of the noise in the numbers.

The Table 2 compares startup times of several programs. As can be seen, the 64-bit system, perhaps surprisingly, is significantly faster in two of them and comparable in bootup times. The Table 3 compares compilation of the package gimp.

As can be seen on Table 4 the memory consumption grows up by about $\frac{1}{4}$ as expected, but due to relative compactness of CISC AMD64 instruction set, the increase is much smaller than one seen after switching to RISC or VLIW systems. In fact Tables 5 and 6 shows decrease in the code section sizes. The major growths can be seen in the section `.eh_frame` that is usually not load into the memory and sections related to the dynamic relocations. According to our benchmarks these are not critical, since dynamic loader is still slightly faster in 64-bit version compared to 32-bit.

Overall, we can recommend use of 64-bit system instead of 32-bit on AMD64 machines intended for desktop use as long as memory consumption increased by 25% is not major limitation (that is hardly the case for computers sold today).

Table 4: Memory Resources Consumption

test	32-bit	64-bit	increase
konqueror	14 M	18 M	28%
gimp	8.6 M	9.9 M	15%
mozilla	22 M	27 M	22%

Table 5: Size of Common Binaries in `/usr/bin`

section	32-bit	64-bit	increase
`.text`	56216 K	53419 K	−5%
`.bss`	18169 K	21098 K	16%
`.data`	10239 K	14076 K	37%
`.rodata`	17543 K	19734 K	12%
`.eh_frame`	546 K	8269 K	1414%
`.rela.plt`	358 K	1076 K	200%
`.rela.dyn`	40 K	126 K	215%
total	80435 K	91141 K	13%

Table 6: Size of Common Shared Libraries

section	32-bit	64-bit	increase
`.text`	71967 K	67526 K	−7%
`.bss`	33463 K	11557 K	−72%
`.dynstr`	13608 K	13587 K	−1%
`.rodata`	12119 K	12217 K	0%
`.dynsym`	11424 K	7611 K	66%
`.eh_frame`	6367 K	12730 K	99%
`.data`	6018 K	9695 K	61%
`.rela.dyn`	4382 K	12844 K	193%
`.plt`	3898 K	6499 K	66%
`.rela.plt`	1293 K	3888 K	200%
`.got`	823 K	1654 K	100%
total	171812 K	198111 K	15%

5 Runtime Library Optimizations

We made following optimizations to glibc:

- Assembly optimized math functions
- Assembly optimized `memcpy` and `memset` functions that do use prefetch and streaming moves for large blocks
- We found `malloc` implementation in glibc 2.2 to be bottleneck. `malloc` in glibc 2.3 solves this problem.

6 Conclusion

The performance of 64-bit code produced by GCC is superior to 32-bit for CPU bound integer and numeric programs (even in comparison to the best optimizing 32-bit compilers available).

Most important optimizations include usage of newly available extended registers, register argument passing conventions, use of SSE for scalar floating point computations and relaxed stack frame layout restrictions by using DWARF2 unwind information for stack unwinding. The code section of 64-bit binaries is, on the average, 5% smaller than code section of 32-bit binary.

Most noticeable problem is the growth of data structures caused by 64-bit pointers. This problem is noticeable as regression in mcf, parser and gap SPEC2000 benchmarks as well as about 25% increase in memory overhead of usual desktop applications and 10% increase of executable file sizes.

Despite that the overall system performance seems to be improved even for (nontrivial) benchmarks targeted to measure extra overhead of increased memory bandwidth, such as program startup times (0%–20% speedup), compilation (12%) or SPEC2000 integer benchmark suite (3.3%). Still it can be worthwhile to implement LP32 code model to provide an alternative for memory bound applications.

The aggressive optimizations in argument passing conventions also brought several compatibility problems especially when dealing with variable argument lists. Other common problem is lack of support for DWARF2 in `gas` assembler making use of assembly functions in AMD64 code difficult.

By eliminating the common bottleneck of IA-32 code (such common memory accesses caused by register starve ISA and argument passing conventions), the code became more sensitive to compiler optimizations. Number of optimizations we evaluated are more effective in 64-bit than on 32-bit especially those improving instruction decoding bandwidth (AMD64 code usually consists of more instructions with shorter overall latency), instruction scheduling and those that increase register pressure.

In comparison to DEC Alpha EV56 architecture, AMD Opteron is considerably less sensitive on instruction scheduling and in-lining. The first is caused by out-of-order architecture and the second probably by smaller L1 cache.

7 Acknowledgements

The port of GCC to AMD64 was done by a team of developers and I'd like to acknowledge their contributions. Without the numerous discussions and the joint development the port and therefore this paper would not be possible.

Geert Bosch designed stack unwinding and exception handling ABI. Richard Brunner, Alex Dreyzen and Evandro Menezes provided a lot of help in understanding the AMD Opteron hardware. Zdeněk Dvořák implemented the new loop unrolling pass, improved DWARF2 support and did number of improvements to profile based optimizations framework. Andrew Haley finished the gcj (Java compiler) port started by Bo Thorsen. Richard Henderson reviewed majority of the GCC changes. Jan Hubička implemented the first versions of GCC and Binutils ports, co-edited ABI document, realized the AMD Opteron specific optimizations and some generic ones (unit at a time mode, profile feedback optimizations framework, tracer). Andreas Jaeger ported glibc, provided SPEC2000 testing framework, co-edited ABI document and fixed number of GCC and Binutils bugs. Jakub Jelínek designed and implemented the thread local storage ABI. Michal Ludvig and Jiří Šmíd realized the GDB port. Michael Matz worked on the new register allocator and fixed plenty of GCC bugs. Mark Mitchell edited the ABI document and set up WWW and CVS of the project. Andreas Schwab and Bo Thorsen fixed number of problems in the linker and assembler. Josef Zlomek redesigned the basic block reordering pass and fixed number of bugs in GCC.

Andreas Jaeger and Evandro Menezes also reviewed the paper and helped to clarify it.

References

[AMD64] *AMD x86-64 Architecture Programmer's Manual*, AMD (2003).

[Opteron] *Software Optimization Guide for the AMD Opteron™ Processor*, AMD (2003).

[Pentium4] *IA-32 Intel Architecture Optimization Manual*, Intel (2003).

[AMD64-PSABI] *UNIX System V Application Binary Interface; AMD64 Architecture Processor Supplement, Draft*, (Ed. J. Hubička, A. Jaeger, M. Mitchell), `http://www.x86-64.org`, (2003)

[i386-ABI] *UNIX System V Application Binary Interface; IA-32 Architecture Processor Supplement*, Intel (2000).

[IA-64-ABI] *UNIX System V Application Binary Interface; IA-64 Processor ABI Supplement*, Intel (2000).

[PPC-ABI] *UNIX System V Application Binary Interface; PowerPC Processor ABI Supplement* (1995).

[DWARF2] *DWARF Debugging Information Format, Version 2.0.0* UNIX International, Program Languages SIG (1993).

[FDO] *Design and Analysis of Profile-Based Optimization in Compaq's Compilation Tools for Alpha*, Journal of Instruction-Level Parallelism 3 (2000), p. 1–25.

[STC] A. Ramirez, J. L. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero, *Software trace cache*, Proc. 13th Intl. Conf. on Supercomputing (1999), p. 119–126.

[profile] Y. Wu and J. R. Larus, *Static branch frequency and program profile analysis*, In Proceedings of the 27th International Symposium on Microarchitecture (1994), p. 1–11.

Table 7: 64-bit SPECint 2000 with Standard Optimization (AMD Opteron)

Performance (relative speedups in percents):

options	gzip	vpr	gcc	mcf	crafty	parser	eon	perl	gap	vortex	bzip2	twolf	avg
	1.32	0.14	−0.45	−0.45	−0.17	0.19	0.41	0.11	0.60	0.28	0.27	−0.54	0.13
standard optimization	105.37	82.29	90.55	12.06	87.14	58.23	451.70	97.05	101.18	75.30	142.14	55.99	93.40
`-fguess-branch probabilities`	4.40	4.45	2.90	0.00	2.73	0.19	5.58	5.96	7.43	21.60	2.56	−1.46	4.10
`-fschedule-insns2`	1.62	1.44	2.40	0.22	0.32	0.78	4.90	1.28	−0.45	4.34	0.41	0.93	1.46
`-fstrict-aliasing`	1.48	4.62	1.93	0.00	3.68	0.58	−2.34	1.75	0.75	4.27	4.79	−2.34	1.19
`-mfpmath=sse`	1.93	3.98	−0.23	0.00	−0.09	−0.39	2.11	0.00	1.81	3.94	0.27	0.80	1.06
prologue using move	−0.74	0.14	0.34	0.00	4.04	0.98	−0.43	1.43	0.30	5.71	−0.28	0.13	0.93
full sized loads and moves	−1.76	−0.29	−0.46	0.88	0.96	−0.20	24.90	−1.52	−0.45	−1.04	0.97	−3.61	0.93
`-fgcse`	1.17	4.28	−1.77	1.35	0.48	1.38	2.33	1.75	−1.48	1.55	1.26	0.13	0.92
`-foptimize sibling-calls`	1.62	0.43	−0.12	0.00	3.33	0.00	2.33	−0.35	1.51	2.44	0.27	0.26	0.92
`-finline-functions`	1.62	0.71	0.22	1.11	0.32	3.08	0.30	−1.04	0.58	−0.99	2.21	0.67	0.65
`-fomit-frame-pointer`	0.29	1.58	0.56	0.67	5.00	1.57	−3.03	3.07	−0.60	0.47	2.41	−3.48	0.39
`-freorder-blocks`	3.61	−0.29	−0.57	0.22	2.31	−0.78	0.72	4.06	0.75	3.45	1.84	−5.31	0.39
`-maccumulate-outgoing-args`	1.92	−0.58	0.78	0.45	0.24	−0.39	1.04	−0.12	−0.60	−1.13	0.13	0.80	0.26
`-mred-zone`	1.47	0.14	1.35	−0.23	1.30	−0.20	−1.73	0.00	−0.30	−0.29	0.55	−0.67	0.13
partial SSE moves	−0.30	5.89	−0.92	0.00	0.07	0.00	−1.17	0.00	0.00	−0.10	−0.14	−3.36	−0.27
aggressive optimization	6.34	4.97	8.81	0.67	1.29	25.43	24.14	12.29	7.51	5.69	5.42	4.65	8.40
`-fbranch-probabilities`	5.95	1.71	7.13	0.22	−0.65	16.76	2.98	3.90	0.14	6.95	0.27	3.73	4.07
`-funroll-all-loops`	4.16	0.42	5.60	0.00	−4.28	0.77	16.42	4.02	1.35	0.57	1.82	1.46	2.50
`-funroll-loops`	3.71	0.28	4.17	0.00	0.08	0.58	15.35	1.61	1.35	−4.78	0.55	3.32	2.23
all prologue using move	−0.60	0.56	2.38	−0.23	−0.40	0.58	3.73	3.19	−0.15	−4.29	0.55	4.68	1.05
`-ffast-math`	1.78	0.28	0.67	0.00	−0.25	−0.20	0.31	−0.81	0.15	2.67	1.12	2.64	0.78
`-frename-registers`	−0.15	0.56	−0.68	0.00	0.08	0.58	1.34	−2.19	−0.76	−1.25	0.97	4.92	0.65
`-funit-at-a-time`	0.89	2.71	0.79	0.45	0.72	0.38	0.00	−0.47	−0.45	0.68	0.69	−0.93	0.39
`-ftracer`	3.12	0.14	1.57	0.00	1.13	−0.20	1.76	0.91	−7.81	−3.83	1.40	2.40	0.13
`-cmodel=medium`	−4.30	−1.00	−0.45	0.00	−10.84	0.00	2.18	−3.57	−5.83	−6.27	−2.23	−0.27	−2.51
`-fpic`	−9.11	−1.72	−1.68	0.89	−18.21	−0.78	−1.36	−16.79	−3.76	−15.16	−6.18	−1.48	−6.20

File size (relative increase of the size of stripped binaries in percents):

options	gzip	vpr	gcc	mcf	crafty	parser	eon	perl	gap	vortex	bzip2	twolf	total
standard optimization	−11.24	−23.04	−23.74	−20.59	−17.13	−13.77	−13.71	−20.00	−36.54	−9.42	−15.83	−39.29	−22.31
`-maccumulate-outgoing-args`	−0.42	−4.02	−3.47	−3.34	−0.35	−3.30	−3.15	−3.29	−4.31	−3.60	5.16	−2.51	−3.25
`-fomit-frame-pointer`	−0.26	1.72	−1.13	−0.20	0.04	−3.76	−1.94	−1.24	−1.07	2.08	−0.08	−0.99	−0.71
`-fstrict-aliasing`	0.00	−0.68	−0.15	0.00	0.00	0.00	0.22	0.00	−0.34	−0.66	0.00	−5.02	−0.40
`-mred-zone`	0.00	−0.11	−0.19	0.00	−0.02	0.00	−0.76	0.59	−0.02	0.00	0.00	−0.04	−0.09
`-fschedule-insns2`	0.00	0.02	−0.15	0.00	0.01	0.00	0.02	0.00	0.00	0.02	0.00	−0.07	−0.05
`-fgcse`	−0.11	0.04	−0.16	0.19	0.03	0.11	0.44	0.68	−0.01	−0.68	0.00	−1.16	−0.05
`-foptimize sibling-calls`	0.00	−0.03	0.08	0.00	−0.02	0.00	−0.76	0.48	−0.16	−0.01	−0.23	−0.10	−0.03
partial SSE moves	0.00	0.00	0.00	0.00	0.00	0.00	0.16	0.00	0.00	0.00	0.00	0.01	0.02
full sized loads and moves	0.00	0.00	0.04	0.00	1.21	0.00	0.00	0.00	0.08	−0.01	0.00	0.11	0.08
`-mfpmath=sse`	0.00	−0.64	−0.15	0.00	0.00	0.00	2.34	−0.01	0.00	0.00	0.00	−1.64	0.13
prologue using move	−0.11	1.06	1.01	0.00	1.26	−0.34	0.91	0.84	1.44	2.55	0.00	0.16	1.14
`-freorder-blocks`	7.06	2.71	4.43	0.00	4.05	3.67	1.07	5.72	3.42	5.60	10.89	4.22	4.19
`-finline-functions`	−0.73	1.15	8.85	−0.20	0.24	28.60	0.12	6.55	3.37	1.99	29.84	0.68	5.49
`-fguess-branch probabilities`	7.00	4.41	5.82	0.00	3.60	3.34	2.64	6.67	5.85	8.74	10.89	3.97	5.66
`-fasynchronous unwind-tables`	7.12	10.28	7.38	6.31	3.76	17.16	4.83	9.26	9.04	7.88	18.14	5.34	7.71
`-fbranch-probabilities`	−4.91	−2.07	−2.20	0.82	0.11	0.02	−2.44	−3.92	−3.74	−4.72	−7.30	−1.80	−2.85
`-funit-at-a-time`	−22.64	−4.95	−1.50	0.00	0.00	0.00	0.00	−0.82	−0.08	−0.01	0.00	−0.10	−1.09
`-ffast-math`	0.00	−0.03	0.00	0.00	0.00	0.00	0.00	−0.68	0.00	−0.02	0.00	0.01	−0.09
`-frename-registers`	0.00	0.26	0.97	0.00	0.28	0.00	1.99	0.68	0.24	0.04	0.00	1.83	0.78
all prologue using move	−0.73	4.14	1.14	−0.96	−0.33	2.18	1.35	0.87	1.60	0.52	−0.77	2.38	1.17
`-ftracer`	0.00	1.27	1.29	0.00	0.13	0.00	2.50	2.01	2.46	1.31	0.00	1.54	1.56
`-funroll-loops`	13.30	7.92	3.18	1.34	4.22	7.11	1.26	2.70	12.57	0.02	9.82	8.70	4.21
`-funroll-all-loops`	13.30	9.53	4.29	24.50	4.71	14.20	1.43	3.38	15.76	0.66	9.82	14.40	5.71
`-fpic`	12.11	6.53	3.62	1.14	21.40	9.38	1.92	6.48	15.53	9.16	7.06	16.66	7.55
`-mcmodel=medium`	13.62	8.10	7.10	0.00	17.57	7.44	6.35	8.29	8.35	6.64	9.90	13.33	8.09
aggressive optimization	−14.42	4.03	21.89	5.12	6.44	44.45	−0.47	8.80	7.38	0.73	40.05	3.93	11.08

Table 8: 64-bit SPECfp 2000 with Standard Optimization (AMD Opteron)

Performance (relative speedups in percents):

options	wupwise	swim	mgrid	applu	mesa	art	equake	ammp	sixtrack	apsi	avg
	−0.28	−0.13	0.00	0.00	0.23	−2.07	0.14	0.00	0.00	0.00	−0.16
standard optimization	102.22	54.49	633.14	220.37	79.20	22.69	90.76	111.08	204.34	192.64	142.52
`-mfpmath=sse`	9.30	0.12	3.31	2.38	11.68	102.55	0.28	8.32	11.53	6.01	12.43
`-fguess-branch-probabilities`	7.62	0.00	6.42	2.78	7.48	0.42	−2.23	−1.27	−0.29	4.72	2.75
partial SSE moves	2.86	0.13	2.95	3.21	3.34	−3.26	0.86	3.11	3.86	3.33	2.12
full sized loads and moves	2.13	0.26	1.35	1.98	6.38	0.69	0.00	2.00	1.45	1.55	1.78
`-fstrict-aliasing`	0.00	0.12	0.00	0.19	2.22	5.22	−2.23	0.90	0.00	5.08	1.44
`-fschedule-insns2`	2.23	0.00	7.72	0.78	0.34	−1.40	−2.50	0.90	4.50	1.01	1.28
`-freorder-blocks`	0.97	0.12	0.18	0.19	13.09	2.28	0.28	0.00	−1.42	0.00	1.28
`-fomit-frame-pointer`	2.51	0.00	4.53	0.38	−0.58	−1.80	−1.13	0.90	−0.29	3.63	0.95
prologue using move	−3.24	0.00	0.00	0.00	3.58	0.69	0.00	−0.14	0.00	0.00	0.15
`-finline-functions`	0.13	0.12	0.00	0.19	1.85	−1.51	1.84	−0.52	0.28	−0.17	0.15
`-foptimize sibling-calls`	0.82	0.12	0.18	0.19	−0.46	−0.97	0.00	0.12	0.00	0.00	0.00
`-mred-zone`	0.00	0.00	0.00	0.38	0.57	0.97	−2.10	−0.26	0.00	0.16	0.00
`-maccumulate-outgoing-args`	0.55	−0.13	0.18	0.00	0.45	−3.46	0.00	0.00	−0.29	0.33	−0.16
`-fgcse`	1.37	0.00	−7.19	−5.15	−0.23	0.69	0.42	−0.64	−4.14	−2.13	−1.71
aggressive optimization	5.57	−0.91	6.60	4.26	4.14	−1.93	7.96	3.58	10.63	−2.34	3.15
`-funroll-all-loops`	2.72	−0.13	1.88	2.32	−1.50	5.58	0.42	3.58	−0.29	1.16	1.58
`-funroll-loops`	2.72	0.00	1.88	2.51	−0.92	2.67	2.13	3.58	−0.29	1.16	1.57
`-ffast-math`	0.81	0.00	0.00	2.13	1.26	−3.16	0.99	4.74	0.57	1.50	0.94
all prologue using move	4.18	0.00	−0.39	0.19	0.23	−0.98	1.86	−0.27	1.14	0.34	0.63
`-fbranch-probabilities`	−3.44	0.12	−0.94	0.38	15.14	−1.40	−0.15	−0.65	0.85	−3.35	0.15
`-funit-at-a-time`	0.13	0.12	−0.19	0.00	3.93	−3.54	0.14	0.12	0.00	−0.17	0.15
`-frename-registers`	−3.54	−0.26	5.66	−0.39	−7.23	−1.11	4.97	3.46	0.86	−0.34	0.15
`-ftracer`	−0.82	0.00	0.00	0.00	−2.87	−2.35	−0.15	0.77	0.86	−0.67	−0.64
`-cmodel=medium`	2.73	−0.26	−0.19	−0.39	−3.69	−0.83	−0.72	−1.03	−14.95	−0.17	−1.90
`-fpic`	0.95	0.00	0.37	−0.97	1.72	−0.29	0.71	−0.13	−20.98	−0.17	−1.90

File size (relative increase of the size of stripped binaries in percents):

options	wupwise	swim	mgrid	applu	mesa	art	equake	ammp	sixtrack	apsi	total
standard optimization	−25.71	−26.52	−36.03	−60.14	−34.62	−15.82	−33.14	−32.33	−38.32	−30.33	−36.85
`-maccumulate-outgoing-args`	−1.63	−0.71	−1.83	−0.71	−3.40	−2.07	−1.80	−2.77	−1.12	−1.17	−1.89
`-fschedule-insns2`	0.00	0.00	0.00	0.05	0.00	0.00	0.00	0.02	−0.43	0.00	−0.21
`-mred-zone`	0.00	0.00	−0.19	−2.31	−0.13	−0.08	−0.14	−0.12	−0.03	−0.12	−0.14
`-fgcse`	0.00	−8.64	−4.00	−10.19	−0.74	1.91	−0.38	0.00	1.70	−3.61	−0.07
`-fstrict-aliasing`	0.00	0.00	0.00	0.00	−0.13	0.07	0.00	−0.05	0.00	0.00	−0.04
`-foptimize sibling-calls`	0.00	0.00	0.00	0.00	−0.24	0.00	0.00	0.04	−0.02	0.68	−0.02
full sized loads and moves	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.40	0.00	0.75	0.08
`-fomit-frame-pointer`	0.00	0.47	0.75	−1.97	−0.05	0.39	−0.14	0.37	0.12	5.74	0.43
partial SSE moves	0.00	0.23	0.00	0.71	0.79	0.00	0.00	0.24	0.43	0.89	0.53
prologue using move	−0.28	0.00	0.00	0.11	1.78	0.00	0.00	0.26	−0.02	0.70	0.53
`-freorder-blocks`	0.00	0.47	0.00	0.11	2.44	0.00	0.00	2.62	0.86	1.37	1.38
`-mfpmath=sse`	0.00	2.16	0.00	6.26	−1.57	0.00	−0.14	3.19	2.65	4.39	1.60
`-fguess-branch probabilities`	−0.28	1.43	0.00	−0.36	5.10	12.16	10.56	3.04	0.41	1.19	2.09
`-finline-functions`	0.00	0.00	0.00	0.00	5.39	19.96	0.13	0.42	1.29	1.50	2.45
`-fasynchronous-unwind-info`	9.34	3.15	6.75	1.92	10.46	16.55	13.01	6.21	1.25	3.83	4.67
`-fbranch-probabilities`	0.64	0.15	0.76	0.19	−5.23	0.70	0.61	−2.11	−0.28	−0.06	−1.58
`-ffast-math`	0.00	−0.95	0.00	0.58	−0.83	−13.04	−0.27	−5.57	0.86	0.00	−0.35
`-funit-at-a-time`	0.00	0.00	0.00	0.00	−0.07	0.00	0.00	−0.03	0.00	0.00	−0.03
all prologue using move	−0.28	1.40	0.37	1.29	0.78	−1.02	−0.40	2.26	0.61	1.96	0.86
`-ftracer`	0.00	0.00	0.00	0.00	2.37	0.07	0.00	5.45	0.43	3.35	1.51
`-frename-registers`	0.00	0.47	0.00	2.65	1.78	0.00	0.00	2.60	2.58	0.86	2.10
`-funroll-loops`	1.93	24.69	6.32	6.42	7.95	20.05	0.65	11.14	3.02	6.63	5.63
`-funroll-all-loops`	1.93	24.69	7.25	6.42	8.19	20.05	2.35	11.14	3.02	6.63	5.73
`-fpic`	0.45	0.23	0.93	2.24	5.92	9.28	7.71	4.91	8.04	3.75	6.51
`-mcmodel=medium`	0.09	4.93	0.00	7.49	3.53	0.85	1.83	5.45	24.62	6.36	14.32
aggressive optimization	71.81	164.20	125.37	57.30	11.28	97.53	52.54	12.91	26.21	34.10	26.45

Table 9: 64-bit SPECint 2000 with Aggressive Optimization (AMD Opteron)

Performance (relative speedups in percents):

options	gzip	vpr	gcc	mcf	crafty	parser	eon	perl	gap	vortex	bzip2	twolf	avg
	−0.28	−0.41	0.20	−0.45	0.00	−0.16	0.00	−0.11	0.84	0.00	0.13	0.38	0.12
aggressive optimization	112.35	91.73	103.60	14.72	86.01	97.56	589.65	130.46	111.79	74.46	151.98	56.79	106.81
`-fbranch-probabilities -fguess-branch...`	8.40	2.62	10.71	0.22	3.38	21.72	27.67	27.67	14.24	10.37	4.39	−1.56	9.49
full sized loads and moves	1.00	0.67	−0.53	0.00	0.97	−0.48	56.39	1.79	0.71	0.62	0.13	4.64	4.61
`-fbranch-probabilities`	2.69	0.00	5.62	−0.45	2.62	19.85	−0.92	11.94	4.06	2.29	1.07	0.51	3.77
`-m64`	9.90	0.27	3.39	−22.19	42.29	−2.13	45.66	0.30	−1.25	6.29	8.28	−13.33	3.38
`-funroll-loops`	1.69	0.54	0.41	0.22	0.88	1.41	16.94	7.59	0.56	1.73	0.93	4.62	3.12
`-freorder-blocks`	4.95	1.22	4.51	0.22	3.89	1.89	2.40	13.06	−0.56	−1.42	0.40	1.15	2.48
`-fomit-frame-pointer`	0.13	0.00	2.19	0.44	2.03	1.73	2.31	5.38	−0.28	1.08	1.47	5.05	2.10
`-fstrict-aliasing`	−0.56	4.80	0.82	0.44	1.04	1.89	1.61	2.08	1.72	1.64	5.88	1.15	1.85
`-finline-functions`	−0.42	0.54	1.55	2.02	1.86	5.21	1.01	−0.31	0.42	3.62	3.13	2.75	1.85
`-ftracer`	−0.69	−0.27	0.30	0.00	1.12	0.78	5.20	3.93	0.14	0.27	0.53	4.90	1.60
`-fschedule-insns2`	0.27	2.62	0.41	0.22	4.24	0.46	2.57	1.55	0.99	3.34	1.61	0.64	1.47
`-mred-zone`	−0.42	0.13	0.61	0.66	0.96	0.31	−1.33	1.56	−0.56	7.01	−0.14	3.56	1.22
`-fgcse`	2.70	4.06	1.14	−0.23	3.47	−0.77	−0.51	−0.82	2.29	1.27	0.93	0.25	1.10
`-mfpmath=sse`	−0.28	2.48	−0.52	0.66	1.95	0.78	9.05	0.72	0.14	−2.80	−0.14	1.42	1.10
`-frename-registers`	−0.42	1.22	−1.13	−0.45	4.24	0.46	−1.90	−0.72	−0.97	1.91	1.47	4.81	0.98
`-funit-at-a-time`	−0.56	3.50	−1.23	0.22	1.12	0.93	0.16	−1.42	2.73	3.43	−0.27	2.64	0.98
prologue using move	−0.43	0.54	1.06	0.43	1.06	0.79	−2.75	1.89	3.63	6.29	−0.14	−0.26	0.86
partial SSE moves	−0.29	0.81	0.10	−0.44	0.00	0.63	0.00	0.62	0.00	0.26	−0.40	4.78	0.73
`-foptimize sibling-calls`	0.00	−0.14	0.61	0.22	0.96	0.78	1.96	0.00	−1.93	−1.86	−0.27	3.15	0.60
`-maccumulate-outgoing-args`	−0.28	0.94	−0.11	−0.23	2.53	0.46	1.18	−0.72	2.43	−0.81	0.13	0.63	0.48
`-fstrength-reduce`	−0.42	0.26	−1.22	0.00	0.64	0.00	−0.59	−1.81	0.42	4.30	−0.14	−0.13	0.00
all prologue using move	−1.13	−0.27	−0.32	−0.22	1.28	0.94	6.46	−0.11	1.54	−1.33	0.39	0.50	0.61
`-ffast-math`	−0.28	0.40	−1.24	−0.23	−1.92	0.00	0.08	0.10	0.56	1.34	−0.27	−3.56	−0.73
`-fpeel-loops`	0.00	0.13	−1.13	0.22	−1.20	−0.62	0.08	−1.34	−1.69	−3.86	−0.40	−0.26	−0.73
`-funroll-all-loops`	0.00	0.13	0.10	0.00	−0.48	−0.16	−0.84	2.04	−2.12	−5.58	0.26	−7.90	−1.70
`-cmodel=medium`	−5.12	−1.21	−2.97	0.44	−10.61	−0.78	−1.09	0.00	0.28	−4.85	−0.67	−7.74	−3.28
`-fpic`	−12.73	−1.89	−2.36	−0.89	−13.88	−6.96	−4.36	−12.79	−2.11	−18.23	−10.03	−8.87	−8.12

File size (relative increase of the size of stripped binaries in percents):

options	gzip	vpr	gcc	mcf	crafty	parser	eon	perl	gap	vortex	bzip2	twolf	total
aggressive optimization	−24.01	−19.87	−6.95	−16.43	−11.81	24.89	−14.11	−12.48	−31.74	−8.77	17.87	−36.91	−13.57
`-fbranch-probabilities`	−12.51	−8.07	−5.50	−0.95	−2.64	−2.55	−5.80	−7.77	−14.58	−5.56	−12.11	−10.22	−7.10
`-maccumulate-outgoing-args`	−1.79	−1.55	−2.33	−1.44	−0.87	−2.85	−3.31	−1.77	−4.10	−3.78	3.05	−1.85	−2.58
`-fgcse`	0.73	−1.16	−1.95	−0.37	−1.92	−1.27	−0.32	−0.59	−0.38	−0.68	−0.06	−3.33	−1.23
`-fomit-frame-pointer`	−1.38	1.02	−0.81	−0.91	−0.27	−1.20	−1.94	−1.43	−1.10	1.41	−0.06	−1.20	−0.72
`-fstrict-aliasing`	0.12	−1.14	−0.11	−0.73	0.00	0.36	0.36	−0.58	−0.56	−0.66	0.00	−5.14	−0.46
`-mred-zone`	0.00	−0.06	−0.06	0.00	0.00	0.00	−0.34	−0.04	−0.02	0.12	0.00	−0.05	−0.05
`-fschedule-insns2`	−0.07	−0.06	−0.07	−0.19	0.01	0.07	0.00	−0.01	−0.02	0.00	0.00	−0.04	−0.03
`-foptimize sibling-calls`	0.06	−0.04	0.10	0.00	0.00	−0.04	−0.45	−0.20	0.13	−0.01	−0.06	−0.05	−0.03
`-fstrength-reduce`	0.24	0.11	−0.01	0.18	0.01	0.03	−0.02	0.00	0.10	0.00	0.00	0.12	0.02
partial SSE moves	0.00	0.27	0.00	0.00	0.00	0.01	0.24	0.00	0.00	0.00	0.00	0.01	0.03
full sized loads and moves	0.18	0.09	0.17	0.00	0.00	0.40	0.01	0.00	0.13	0.00	0.00	0.07	0.10
`-mfpmath=sse`	0.00	−1.35	−0.05	−0.55	−0.14	−0.08	3.34	−0.58	0.00	0.00	0.00	−1.39	0.15
prologue using move	0.00	0.07	0.14	0.00	−0.05	0.40	−0.02	0.45	0.28	0.37	−0.06	0.06	0.20
`-funroll-loops`	1.73	0.98	0.34	3.97	1.51	3.22	0.28	0.04	1.00	0.00	0.00	0.77	0.52
`-freorder-blocks`	0.24	0.11	1.05	−0.55	0.00	−0.04	0.20	0.63	0.36	0.00	0.00	0.21	0.53
`-frename-registers`	1.35	1.18	1.26	0.00	1.47	0.71	2.27	0.67	0.62	0.66	0.00	2.19	1.16
`-ftracer`	0.67	1.36	1.57	2.61	2.02	2.29	0.44	1.30	1.61	2.01	0.00	0.58	1.43
`-fbranch-probabilities -fguess-branch...`	6.09	4.09	5.60	5.44	6.03	9.87	−0.21	3.90	3.58	4.49	7.78	3.27	4.40
`-funit-at-a-time`	−14.10	2.25	12.02	0.00	2.04	5.62	0.00	4.14	6.08	2.66	7.60	1.92	5.94
`-m64`	16.48	−2.64	8.02	18.47	−19.00	15.52	0.25	11.38	9.65	−5.69	8.64	−3.44	3.90
`-finline-functions`	8.71	7.94	23.54	2.80	3.51	39.11	−0.09	11.96	9.86	4.17	39.65	2.71	12.98
`-ffast-math`	0.00	−0.02	0.03	0.00	0.00	0.00	0.00	−0.05	0.00	−0.02	0.00	0.01	0.00
`-funroll-all-loops`	0.00	0.23	0.04	2.18	0.00	1.26	0.00	0.57	0.09	0.00	0.00	−2.94	0.03
`-fpic`	16.27	4.69	−6.01	0.18	17.87	−21.91	0.96	1.39	6.50	7.12	−21.77	14.97	0.38
`-fpeel-loops`	1.57	0.39	0.35	1.63	1.98	5.80	0.00	0.57	0.96	0.00	0.00	1.25	0.66
all prologue using move	2.18	2.85	1.30	1.45	0.26	2.63	2.31	1.71	2.95	2.77	−0.72	2.62	1.91
`-mcmodel=medium`	14.15	9.85	7.56	19.12	18.58	7.95	5.97	9.93	9.90	7.91	21.15	12.94	9.01

Table 10: 64-bit SPECfp 2000 with Aggressive Optimization (AMD Opteron)

Performance (relative speedups in percents):

options	wupwise	swim	mgrid	applu	mesa	art	equake	ammp	sixtrack	apsi	avg
	1.30	0.00	0.89	0.56	−5.34	−0.28	0.00	−0.13	−1.29	1.21	−0.16
aggressive optimization	101.11	53.87	686.79	225.30	101.38	26.80	100.81	123.51	225.00	180.97	149.23
`-m64`	5.00	−0.27	16.25	9.79	28.55	83.54	−1.31	19.17	28.33	20.86	19.34
`-mfpmath=sse`	13.97	0.12	2.40	2.33	7.04	100.28	1.79	16.64	22.22	5.67	13.80
`-fbranch-probabilities`	−0.83	0.39	10.83	3.96	19.62	2.23	−0.28	6.85	2.24	0.70	3.98
`-fguess-branch...`											
partial SSE moves	1.58	0.13	2.18	1.76	0.70	1.27	−2.51	3.17	6.14	2.54	1.74
`-fstrict-aliasing`	0.13	0.00	0.00	0.00	−0.90	4.49	1.37	5.49	0.00	4.71	1.73
full sized loads and moves	−2.25	0.26	3.31	1.16	4.29	2.40	2.92	0.86	2.25	0.89	1.57
`-fschedule-insns2`	0.13	0.12	13.06	0.57	−9.93	1.53	−0.68	5.49	3.71	1.58	1.41
`-ftracer`	0.27	0.00	−0.19	−0.19	−2.85	0.97	1.79	1.10	0.00	0.34	0.15
`-mred-zone`	−0.95	0.00	−0.19	1.15	−2.32	0.13	1.09	0.00	0.00	0.00	−0.16
prologue using move	−1.53	0.13	−0.18	−0.20	0.91	−0.84	−0.14	0.00	0.00	−0.18	−0.16
`-frename-registers`	0.00	0.00	4.52	−0.76	−12.07	1.83	3.21	1.84	1.39	−1.03	−0.31
`-fbranch-probabilities`	−1.61	0.00	−0.37	−0.57	7.36	−0.83	−0.14	−0.49	0.83	−4.16	−0.32
`-fomit-frame-pointer`	−1.08	0.00	0.54	0.95	−11.17	−0.69	0.68	0.85	0.00	1.94	−0.62
`-finline-functions`	0.00	0.12	−0.19	0.00	−12.12	2.97	1.23	0.36	−0.28	0.00	−0.77
`-maccumulate-outgoing-args`	3.20	−0.13	0.00	−0.19	−9.94	−0.70	0.40	−0.13	−0.28	0.00	−0.78
`-freorder-blocks`	1.08	0.00	−0.19	−0.19	−11.27	1.11	0.13	1.72	0.00	0.00	−0.78
`-funroll-loops`	−2.43	−0.13	0.00	1.34	−11.02	0.83	0.54	3.25	0.00	0.34	−0.78
`-foptimize sibling-calls`	−1.20	0.00	−0.37	0.00	−13.20	0.97	−0.28	−0.49	0.00	0.34	−1.23
`-fstrength-reduce`	−1.85	0.00	−0.37	5.20	−13.15	−0.14	0.95	−0.85	1.39	−2.04	−1.23
`-funit-at-a-time`	−0.96	0.12	−0.19	−0.19	−11.26	0.00	1.09	0.00	0.00	0.00	−1.24
`-fgcse`	−1.46	−0.39	−7.52	−4.36	−12.53	1.26	0.40	−0.13	−1.63	−3.19	−3.02
`-ffast-math`	−2.01	0.00	−0.19	1.13	14.99	−0.70	2.16	1.45	−0.83	2.94	1.86
`-fpeel-loops`	9.94	0.00	−0.19	0.18	0.00	−0.83	−1.22	0.00	0.00	−0.18	0.62
`-funroll-all-loops`	−0.41	0.12	0.00	−0.19	0.00	0.98	−1.49	−0.13	0.00	0.17	−0.16
`-fpic`	5.42	−0.13	0.00	−0.95	14.84	0.55	−1.76	0.00	−20.67	−0.18	−0.63
all prologue using move	−5.90	0.00	−0.89	−0.39	0.20	−0.28	0.54	−0.62	0.00	0.17	−0.78
`-cmodel=medium`	−0.54	−0.13	−0.55	−1.71	9.68	−3.19	−1.76	−3.88	−16.53	−1.22	−2.01

File size (relative increase of the size of stripped binaries in percents):

options	wupwise	swim	mgrid	applu	mesa	art	equake	ammp	sixtrack	apsi	total
aggressive optimization	−16.48	−15.91	−34.31	−57.92	−33.11	8.36	−29.40	−26.61	−36.44	−25.42	−34.22
`-fbranch-probabilities`	0.55	−8.26	−2.73	−3.79	−12.90	−10.98	−9.59	−7.97	−4.00	−7.95	−7.22
`-maccumulate-outgoing-args`	−1.93	−0.62	−1.78	−0.78	−3.49	−0.97	−0.99	−1.92	−0.80	−1.19	−1.67
`-mred-zone`	0.00	−0.21	−0.37	−2.03	−0.77	−0.13	−0.13	−0.03	−0.01	−0.30	−0.30
`-fstrict-aliasing`	0.00	0.00	0.00	0.00	−0.75	6.80	−10.04	0.00	0.00	−0.18	−0.27
`-fgcse`	0.00	−8.64	−4.00	−10.19	−0.74	1.91	−0.38	0.00	1.70	−3.61	−0.07
`-fschedule-insns2`	0.00	0.00	0.00	0.00	−0.10	0.00	0.00	0.00	0.00	0.00	−0.03
prologue using move	−0.09	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.06	0.00
`-foptimize sibling-calls`	0.00	0.00	−0.37	0.00	−0.18	0.00	0.00	0.00	0.36	0.10	0.13
full sized loads and moves	0.00	0.00	0.00	0.00	0.00	0.00	0.12	0.00	0.34	0.08	0.17
`-freorder-blocks`	0.00	0.00	0.00	0.00	0.03	0.24	0.49	0.00	0.42	−0.09	0.21
`-funit-at-a-time`	0.00	0.00	0.00	0.00	0.11	0.12	4.67	1.85	0.00	0.00	0.23
`-fomit-frame-pointer`	8.70	0.82	0.91	−1.92	−0.51	−0.73	−0.38	0.51	0.40	5.13	0.57
`-fstrength-reduce`	0.00	0.00	0.18	−0.51	0.03	0.00	0.12	0.00	1.20	0.12	0.59
partial SSE moves	0.00	0.20	0.18	0.39	0.77	0.60	0.00	0.00	0.82	0.23	0.65
`-ftracer`	11.68	0.41	−1.26	0.00	0.03	0.36	0.87	5.54	0.00	0.92	0.70
`-funroll-loops`	10.37	14.33	2.03	2.81	0.03	6.59	3.06	2.39	0.35	2.96	1.09
`-fbranch-probabilities` `-fguess-branch...`	12.12	15.26	2.69	3.25	0.02	19.33	5.59	8.65	0.43	4.67	1.92
`-frename-registers`	8.99	0.82	0.54	2.99	2.38	1.85	1.76	2.69	2.57	1.58	2.53
`-finline-functions`	0.00	0.00	0.00	0.00	5.92	18.22	4.94	2.41	1.27	1.84	2.75
`-mfpmath=sse`	8.70	2.96	2.03	8.08	−0.75	6.59	3.99	5.54	5.28	5.13	3.72
`-m64`	45.40	201.01	156.05	26.51	17.41	39.81	27.06	23.41	28.79	38.22	28.68
`-ffast-math`	0.00	−0.83	0.00	0.94	−0.85	−6.44	−4.84	−8.23	0.40	−0.18	−0.81
`-funroll-all-loops`	0.00	0.00	0.00	0.00	0.00	0.24	0.61	0.00	0.00	0.00	0.01
`-fpeel-loops`	0.00	0.00	0.00	1.39	0.00	0.36	1.36	0.00	0.00	0.12	0.07
all prologue using move	−0.49	8.82	1.79	1.28	2.22	8.41	0.99	2.15	0.36	4.23	1.49
`-fpic`	0.65	−6.38	2.35	1.11	5.32	−3.71	13.13	2.23	6.58	3.47	5.21
`-mcmodel=medium`	0.00	9.45	2.17	7.98	5.43	10.44	11.27	5.24	23.48	6.72	14.49

Table 11: 32-bit SPECint 2000 with Aggressive Optimization (AMD Opteron)

Performance (relative speedups in percents):

options	gzip	vpr	gcc	mcf	crafty	parser	eon	perl	gap	vortex	bzip2	twolf	avg
	1.06	−0.14	0.42	0.69	0.11	0.00	−0.13	0.20	−0.28	0.85	0.71	3.52	0.75
aggressive optimization	96.74	76.81	73.11	14.74	56.38	83.61	349.45	111.06	98.34	71.82	122.25	67.09	89.12
`-march=i386` to `k8`	5.23	8.41	3.45	0.17	9.02	6.80	82.00	−0.52	0.41	14.78	2.45	8.52	10.08
`-fbranch-probabilities`	8.34	2.37	12.33	1.40	4.25	7.49	17.57	14.35	8.99	12.75	6.47	0.87	7.37
`-fguess-branch...`													
`-fbranch-probabilities`	2.94	0.41	10.33	0.17	2.91	5.43	0.61	8.82	2.41	8.26	6.45	0.77	3.89
`-fomit-frame-pointer`	8.64	1.36	0.84	0.17	2.26	6.51	0.73	0.41	4.58	2.66	6.25	3.78	3.26
`-fgcse`	1.99	1.52	−2.27	−0.69	0.57	−4.36	5.14	8.00	2.67	2.93	1.86	2.98	1.77
`-finline-functions`	0.90	1.96	0.00	2.84	2.91	6.62	1.86	0.82	1.41	3.34	1.87	1.78	2.17
`-ftracer`	0.15	1.94	4.58	−0.52	−0.34	−2.23	3.94	9.70	0.13	1.74	3.05	0.77	1.78
`-fschedule-insns2`	2.30	2.22	2.47	−0.35	2.32	0.15	0.12	1.87	−0.69	2.04	1.73	2.70	1.52
`-funit-at-a-time`	−0.60	8.91	3.47	−0.18	2.55	−1.50	0.12	7.50	−1.10	1.83	0.28	−0.67	1.39
`-freorder-blocks`	1.99	0.68	7.88	−0.87	3.52	0.76	−0.37	1.24	−0.83	2.23	2.01	−1.00	1.26
`-funroll-loops`	−0.31	−0.55	0.00	0.34	0.22	−1.79	6.77	0.72	0.69	2.71	1.14	3.53	1.25
`-march=ppro` to `k8`	5.91	−1.89	2.37	0.34	0.45	−4.22	2.63	0.30	1.11	0.38	2.75	2.60	1.13
`-maccumulate-outgoing-args`	0.60	−0.28	0.53	0.00	2.67	−2.08	5.95	2.62	0.27	4.06	1.00	−2.15	0.88
`-frename-registers`	−0.30	1.65	−0.94	−1.04	0.68	−2.67	−1.57	0.00	−0.14	2.74	0.85	5.49	0.75
`-foptimize sibling-calls`	−0.16	0.27	2.24	0.34	−0.34	−1.93	−1.21	−0.11	0.69	1.93	0.56	0.11	0.25
`-fstrict-aliasing`	1.07	−1.37	0.21	1.39	−0.12	0.00	0.12	0.10	0.55	0.09	0.71	0.55	0.25
`-fstrength-reduce`	−0.16	0.54	−0.53	−1.04	0.57	−2.51	0.12	0.00	−1.10	−1.14	0.28	1.10	−0.25
`-funroll-all-loops`	3.10	−0.28	0.31	−0.87	0.11	2.73	0.49	2.98	0.68	1.14	−0.15	1.98	1.00
`-mfpmath=sse`	1.83	2.32	1.28	−1.38	0.11	0.45	0.36	0.51	1.39	0.94	0.85	0.32	0.75
`-ffast-math`	−0.31	1.09	0.63	0.34	−0.46	0.15	0.12	0.72	0.55	0.86	0.42	0.44	0.50
`-fpeel-loops`	2.29	0.00	−0.32	−0.52	0.90	3.17	0.00	0.10	−3.43	−0.29	0.70	−1.20	0.00
`-fpic`	−20.49	−5.64	−17.55	−3.28	−29.60	−28.19	−10.27	−29.75	−23.00	−35.03	−25.65	−17.66	−20.81

File size (relative increase of the size of stripped binaries in percents):

options	gzip	vpr	gcc	mcf	crafty	parser	eon	perl	gap	vortex	bzip2	twolf	total
aggressive optimization	−18.85	−6.25	3.51	−21.10	2.34	33.46	−4.21	−6.83	−22.83	−2.91	33.80	−22.33	−4.05
`-fbranch-probabilities`	−14.82	−8.93	−5.82	0.67	−1.96	−3.46	−5.89	−7.95	−14.56	−3.10	−11.81	−10.11	−6.87
`-fgcse`	1.21	−1.15	−1.23	0.00	2.31	−0.93	0.20	0.52	0.21	0.51	−1.59	−1.60	−0.28
`-foptimize sibling-calls`	0.07	0.11	0.09	0.00	0.07	0.00	−1.44	0.05	0.01	−0.03	−1.18	−0.02	−0.14
`-fstrict-aliasing`	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
`-fstrength-reduce`	0.21	0.09	0.02	0.00	0.05	−0.29	0.03	0.09	0.12	0.00	0.00	−0.19	0.02
`-fschedule-insns2`	−0.15	0.21	−0.07	0.00	−0.07	0.00	1.63	−0.02	−0.04	−0.01	0.00	0.03	0.15
`-march=ppro` to `k8`	−2.15	1.33	−0.40	0.00	−0.36	0.00	5.56	−0.29	−0.49	0.10	−1.18	0.31	0.40
`-funroll-loops`	3.06	0.81	0.32	0.00	1.16	2.91	0.08	0.21	0.88	0.08	2.31	0.31	0.48
`-frename-registers`	0.49	0.48	0.52	0.00	0.51	0.00	1.42	0.81	0.22	0.10	1.02	0.31	0.55
`-freorder-blocks`	−0.08	−0.06	1.22	0.00	0.50	−0.03	0.17	0.82	0.29	0.10	0.53	0.22	0.62
`-fomit-frame-pointer`	−1.77	2.89	0.39	0.00	−0.14	0.77	4.52	−0.79	0.17	2.38	−2.80	−0.11	0.95
`-ftracer`	0.00	1.33	1.78	0.00	4.56	2.91	0.31	2.07	1.71	2.56	0.29	0.31	1.80
`-fbranch-probabilities`	6.98	3.72	6.73	0.67	9.29	9.37	−0.26	4.48	3.81	4.67	6.41	2.35	4.93
`-fguess-branch...`													
`-maccumulate-outgoing-args`	1.29	6.40	6.00	0.00	1.95	2.47	0.38	2.07	4.64	19.88	3.13	4.36	5.87
`-funit-at-a-time`	−11.69	6.01	13.64	0.00	2.27	6.00	0.00	4.45	7.07	2.65	6.53	1.86	6.58
`-march=i386` to `k8`	1.43	9.46	9.78	0.00	3.65	6.00	8.00	4.13	6.70	21.21	4.02	8.63	9.24
`-finline-functions`	10.90	8.91	28.84	0.00	3.79	39.55	0.16	13.26	10.95	4.65	50.44	2.30	14.46
`-ffast-math`	0.00	−0.79	0.01	0.00	−0.02	0.00	0.00	−0.13	0.00	−1.23	0.00	−0.06	−0.21
`-funroll-all-loops`	0.00	0.25	0.05	0.00	0.07	2.83	0.00	0.03	0.07	0.03	1.19	0.21	0.15
`-fpeel-loops`	2.19	1.15	0.39	0.00	2.81	6.13	0.00	0.21	0.88	0.02	1.25	1.61	0.72
`-fpic`	12.59	6.19	−4.89	0.00	14.80	−27.60	10.58	4.43	1.15	1.35	−21.21	9.83	0.84
`-mfpmath=sse`	−0.08	1.15	−0.03	0.00	−0.06	0.00	10.10	0.17	0.00	0.00	1.19	−1.80	1.13

Table 12: 32-bit SPECfp 2000 with Aggressive Optimization (AMD Opteron)

Performance (relative speedups in percents):

options	wupwise	swim	mgrid	applu	mesa	art	equake	ammp	sixtrack	apsi	avg
	0.13	0.00	0.00	−0.21	0.28	2.57	−0.14	0.00	6.00	0.00	0.72
aggressive optimization	77.83	27.22	445.45	148.97	56.22	−30.46	92.25	101.18	122.37	156.08	98.56
`-march=i386` to `k8`	6.02	0.00	2.53	3.17	13.31	1.54	−0.65	1.49	−3.05	2.11	2.41
`-fbranch-probabilities` `-fguess-branch...`	3.49	0.39	4.74	4.28	0.72	1.81	−1.42	7.93	−2.16	0.20	1.66
`-fomit-frame-pointer`	−0.14	0.12	3.49	2.25	9.32	1.02	0.38	0.29	0.00	1.03	1.63
`-march=ppro` to `k8`	8.34	0.00	0.00	−0.82	10.41	−1.50	0.26	−0.59	−0.94	−0.62	1.10
`-fstrength-reduce`	10.13	−0.26	1.46	1.03	−8.02	−1.54	0.13	0.89	−0.32	3.64	0.91
`-funroll-loops`	3.93	0.00	0.00	0.61	−7.65	1.81	0.52	4.62	0.95	−0.21	0.36
`-fstrict-aliasing`	0.00	0.00	0.00	0.00	0.00	−1.27	−0.13	0.14	0.00	0.00	0.00
`-frename-registers`	0.81	0.12	−0.62	0.00	−5.69	−0.52	1.98	−0.15	0.63	0.62	−0.19
`-funit-at-a-time`	0.13	0.00	0.00	0.00	−5.75	0.25	2.25	0.29	0.00	0.00	−0.19
`-ftracer`	1.65	0.00	0.00	0.00	−6.54	0.51	0.39	2.26	−0.32	−0.82	−0.37
`-finline-functions`	0.00	0.00	0.00	0.00	−7.14	3.70	1.85	−0.15	0.00	−0.21	−0.37
`-maccumulate-outgoing-args`	2.20	0.00	0.20	0.20	−6.37	−0.76	−0.40	0.00	0.00	0.41	−0.37
`-foptimize sibling-calls`	−0.27	0.00	0.00	0.00	−6.44	2.84	0.00	0.14	−0.32	0.00	−0.37
`-fschedule-insns2`	−0.54	0.13	1.04	2.72	−6.49	−0.26	−1.67	1.34	−6.48	1.04	−0.72
`-freorder-blocks`	0.68	−0.13	0.20	0.00	−4.78	−1.52	−0.13	1.04	−1.55	−0.62	−0.73
`-fbranch-probabilities`	1.78	0.00	−0.21	−2.80	0.00	−2.53	0.26	−1.17	−0.63	−2.23	−0.91
`-fgcse`	2.21	−0.39	0.20	−2.40	−3.99	2.02	−0.13	−0.59	−10.68	0.20	−1.43
`-mfpmath=sse`	2.43	0.25	3.29	−0.21	12.53	97.20	−0.14	1.47	13.20	3.30	10.14
`-ffast-math`	1.21	0.25	0.00	2.04	3.13	−0.26	3.89	0.58	−0.95	3.09	1.44
`-fpeel-loops`	3.78	0.00	0.00	2.25	0.00	0.51	−0.26	0.00	0.00	0.00	0.54
`-funroll-all-loops`	0.00	0.12	0.00	0.00	0.00	−2.54	−0.26	0.14	0.00	0.00	−0.19
`-fpic`	−5.15	0.25	−3.72	3.46	−0.43	−1.31	−10.15	−2.36	−11.64	−1.45	−3.10

File size (relative increase of the size of stripped binaries in percents):

options	wupwise	swim	mgrid	applu	mesa	art	equake	ammp	sixtrack	apsi	total
aggressive optimization	−3.88	−1.94	−20.88	−25.85	−23.54	14.89	−16.01	−17.99	−17.79	−11.69	−18.60
`-fbranch-probabilities`	0.24	−2.71	0.69	−4.31	−14.27	−7.93	−4.87	−11.72	−4.35	−7.09	−7.78
`-fstrict-aliasing`	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
`-march=ppro` to `-march=k8`	0.00	0.53	0.00	−4.45	1.81	0.77	0.10	0.41	−0.60	0.00	0.04
`-funit-at-a-time`	0.00	0.00	0.00	0.00	0.24	0.00	3.26	0.43	0.00	0.00	0.14
`-freorder-blocks`	0.00	0.00	0.00	0.04	0.15	−0.12	0.43	0.31	0.28	0.00	0.20
`-foptimize sibling-calls`	0.00	0.00	0.00	0.00	0.26	0.00	0.00	0.02	0.23	1.56	0.30
`-frename-registers`	0.00	0.26	0.00	0.04	0.25	0.33	0.65	0.02	0.61	0.00	0.38
`-ftracer`	7.98	0.00	0.00	0.00	0.07	0.44	0.76	5.35	0.00	1.44	0.66
`-funroll-loops`	5.15	6.26	0.00	1.15	0.06	7.76	1.21	0.43	0.07	2.48	0.57
`-fgcse`	−1.84	3.89	0.00	−4.45	−0.79	0.11	0.54	0.09	2.85	−3.17	0.76
`-fbranch-probabilities` `-fguess-branch...`	10.49	6.75	0.69	1.60	−0.55	11.19	2.58	7.22	0.63	3.16	1.38
`-fschedule-insns2`	0.00	0.81	0.00	1.44	0.63	0.66	1.32	3.68	2.53	2.07	1.90
`-fomit-frame-pointer`	2.10	1.60	0.00	4.64	2.24	0.00	0.54	4.52	1.22	9.28	2.41
`-fstrength-reduce`	0.00	−1.33	0.00	31.50	−0.04	−2.69	−0.22	−0.54	4.37	3.07	3.14
`-finline-functions`	0.00	0.00	0.00	0.00	6.28	13.54	6.74	1.95	1.85	3.97	3.23
`-march=i386` to `-march=k8`	7.17	−4.61	0.00	1.44	6.05	0.55	0.87	−0.68	4.19	8.23	4.35
`-maccumulate-outgoing-args`	7.52	1.91	0.00	0.71	3.53	1.23	1.77	0.43	6.49	9.36	5.03
`-ffast-math`	0.00	−0.81	0.00	0.23	−1.37	−31.41	−31.50	−6.71	−0.07	−0.78	−1.89
`-funroll-all-loops`	0.00	0.00	0.00	0.00	0.00	0.11	0.65	0.00	0.00	0.00	0.01
`-fpeel-loops`	0.77	0.00	0.00	0.42	0.00	0.22	1.19	0.00	0.06	0.00	0.08
`-fpic`	4.90	−6.17	0.00	−25.58	9.63	−3.10	2.72	5.98	7.44	−0.10	5.74
`-mfpmath=sse`	4.04	7.23	0.00	10.72	2.53	7.29	8.28	15.12	8.83	6.81	7.33

Table 13: 64-bit SPECint 2000 with Aggressive Optimization (DEC Alpha EV56/600Mhz)

Performance (relative speedups in percents):

options	gzip	vpr	gcc	mcf	crafty	parser	eon	perl	gap	vortex	bzip2	twolf	avg
	0.00	−0.66	0.71	0.00	1.63	0.00	0.60	0.00	8.02	5.84	−0.55	4.72	1.96
aggressive optimization	143.98	77.03	73.26	16.94	105.84	141.75	505.83	119.81	128.84	94.27	180.89	71.33	115.27
`-fschedule-insns2`	16.23	10.00	1.51	2.20	11.18	2.75	20.56	8.84	2.87	3.78	15.38	5.63	8.08
`-fschedule-insns`													
`-funit-at-a-time`	1.42	2.63	3.73	3.67	−2.83	28.33	18.57	16.66	0.00	16.42	3.52	5.51	7.63
`-finline-functions`	5.18	2.63	2.18	1.47	14.63	31.62	1.19	8.33	0.00	22.13	4.73	2.70	6.84
`-fbranch-probabilities`	1.45	7.58	6.06	2.22	15.47	27.50	5.03	14.00	2.08	−3.48	0.00	0.65	6.16
`-fguess-branch...`													
`-fbranch-probabilities`	9.30	2.66	6.81	5.97	−4.33	29.66	−1.18	9.93	4.22	17.51	2.31	−0.66	5.44
`-fschedule-insns2`	7.87	6.16	3.75	0.72	7.69	−0.89	7.84	7.38	1.42	3.14	7.89	5.63	5.03
`-fomit-frame-pointer`	0.00	0.00	2.94	0.00	5.34	2.01	5.76	7.69	3.52	3.18	5.26	1.33	2.63
`-freorder-blocks`	0.71	0.00	2.15	0.00	14.10	1.31	−6.94	5.62	3.52	4.48	2.22	2.64	2.63
`-fgcse`	5.42	0.00	1.41	0.72	−1.02	−0.65	14.86	1.19	2.09	2.54	2.82	−0.66	1.94
`-fif-conversion`	2.96	5.47	0.00	2.20	4.97	0.65	13.15	0.00	2.08	3.20	−0.56	1.31	2.61
`-fstrength-reduce`	−3.53	−1.28	1.44	2.18	−3.30	−0.65	22.30	−2.96	2.08	−1.87	2.27	4.08	1.97
`-funroll-loops`	−1.42	0.00	2.18	0.00	22.29	0.00	3.65	−0.60	−1.37	1.87	0.00	−3.88	1.30
`-fstrict-aliasing`	−2.88	4.08	−0.71	0.73	2.13	8.45	−16.97	4.34	4.25	3.16	4.59	0.65	0.65
`-frename-registers`	0.71	0.64	0.71	−0.72	5.40	0.66	5.73	2.40	0.68	−12.50	−1.11	3.44	0.65
`-foptimize sibling-calls`	−2.12	0.00	0.71	−1.42	−14.80	−0.65	2.42	−2.49	0.68	1.86	1.11	−0.65	−1.28
`-ftracer`	0.00	−4.55	0.00	−2.16	−12.07	−0.65	3.06	−2.95	0.00	1.25	1.11	−7.10	−2.59
`-ffast-math`	−1.44	−3.73	−2.12	2.18	7.65	0.00	−1.78	−0.59	1.37	8.60	−0.55	1.32	1.29
`-funroll-all-loops`	0.70	−0.65	−2.78	0.72	2.59	1.30	5.16	4.40	0.00	−3.04	0.55	−3.25	0.00
`-fpeel-loops`	0.00	3.28	−0.71	−1.43	4.44	1.30	−3.51	−2.36	0.00	0.61	0.00	−1.30	0.00
`-fold-unroll-loops`	0.00	0.64	0.00	0.72	−4.62	1.31	10.71	3.03	−1.37	−2.54	−6.63	0.00	0.00
`-fpic`	0.00	−2.64	0.00	0.73	−13.23	3.63	−4.10	−0.65	−1.40	−3.71	5.48	−2.65	−2.05

File size (relative increase of the size of stripped binaries in percents):

options	gzip	vpr	gcc	mcf	crafty	parser	eon	perl	gap	vortex	bzip2	twolf	total
aggressive optimization	−38.22	−29.20	−9.28	−42.75	−28.90	5.66	−49.91	−12.38	−36.23	−17.64	−3.00	−39.40	−22.85
`-fbranch-probabilities`	−10.66	−1.50	−2.43	0.79	−0.71	2.11	−4.12	−6.17	0.00	−3.29	−9.80	−5.73	−3.09
`-fomit-frame-pointer`	−10.98	−3.61	−1.53	0.00	−1.19	−3.23	−7.01	−2.35	−2.88	−2.10	−1.09	−3.01	−2.64
`-fgcse`	−0.25	−1.53	−1.07	0.00	−0.87	−1.56	−1.29	−0.48	0.08	0.01	−10.13	0.00	−0.84
`-fstrict-aliasing`	0.03	−1.22	0.00	0.00	−0.07	−0.28	0.26	−0.20	−0.53	−0.26	0.00	−3.01	−0.28
`-freorder-blocks`	−0.04	0.01	0.31	0.00	−0.43	0.01	−1.35	−0.23	−0.27	0.00	0.00	0.00	−0.09
`-foptimize sibling-calls`	0.06	−0.01	0.23	0.00	0.00	0.01	−1.26	−0.04	0.10	0.00	0.00	0.00	−0.02
`-frename-registers`	0.06	−0.09	0.00	0.00	−0.10	0.02	0.08	−0.09	0.01	−0.03	0.00	0.00	−0.02
`-fif-conversion`	−0.10	−0.19	0.28	0.00	0.15	−0.21	−1.31	0.05	0.04	0.00	0.00	0.00	−0.01
`-fstrength-reduce`	0.06	0.33	0.00	0.00	0.23	−0.48	0.05	0.06	0.20	0.01	0.01	0.00	0.04
`-funroll-loops`	0.06	0.00	0.27	0.00	0.12	0.34	0.00	0.05	0.00	0.00	0.00	0.00	0.12
`-ftracer`	0.04	0.63	2.22	0.00	3.66	5.31	0.11	2.09	−0.11	3.25	0.00	3.19	1.99
`-funit-at-a-time`	−20.22	0.29	9.22	0.83	1.09	6.54	−4.12	4.22	0.00	−1.08	0.30	−2.99	3.12
`-fbranch-probabilities`	0.46	4.61	5.48	0.79	5.40	6.52	0.20	4.37	0.06	4.34	0.42	3.34	3.90
`-fguess-branch...`													
`-fschedule-insns2`	0.00	4.24	4.73	0.00	3.87	0.00	5.63	3.53	4.29	3.47	0.00	3.41	4.06
`-fschedule-insns2`	0.00	4.42	5.01	0.00	3.87	0.00	7.14	4.76	5.25	4.69	0.00	3.41	4.76
`-fschedule-insns`													
`-finline-functions`	0.47	8.20	23.93	0.79	3.89	43.62	−4.17	14.35	0.00	2.22	52.11	−2.89	11.68
`-ffast-math`	−0.31	−0.09	−0.01	−0.40	−0.06	−0.12	−0.04	−0.01	−0.07	−0.04	−0.12	−0.03	−0.04
`-funroll-all-loops`	0.99	0.31	0.00	0.00	0.43	2.29	0.00	0.11	0.00	0.02	0.00	0.00	0.13
`-fpeel-loops`	12.32	0.57	0.03	0.00	2.11	6.22	0.00	0.18	0.00	0.04	0.22	0.00	0.49
`-fpic`	−1.53	1.09	0.12	0.39	1.78	5.18	2.52	1.25	1.35	0.21	1.28	0.80	0.92
`-fold-unroll-loops`	12.39	8.85	−1.48	0.00	5.54	5.61	2.90	2.75	13.59	0.00	11.26	9.30	2.83

Table 14: 64-bit SPECfp 2000 with Aggressive Optimization (DEC Alpha EV56/600Mhz)
Performance (relative speedups in percents):

options	wupwise	swim	mgrid	applu	mesa	art	equake	ammp	apsi	avg
	0.00	−0.75	−0.21	0.00	0.93	0.00	0.83	−0.84	−1.76	0.00
-fschedule-insns2	14.49	10.74	50.22	17.06	28.57	7.60	17.08	24.61	25.41	21.69
-fschedule-insns -fschedule-insns2	1.93	0.00	0.92	3.25	34.50	7.73	4.67	5.26	0.00	5.78
-fstrength-reduce	9.27	0.75	2.71	4.88	2.85	1.19	2.56	0.84	1.81	3.17
-fbranch-probabilities -fguess-branch...	3.12	0.00	1.44	1.33	14.21	7.10	3.41	−0.83	0.90	3.14
-ftracer	1.85	0.00	1.02	0.20	8.54	1.14	0.82	0.84	6.79	2.36
-fbranch-probabilities	1.85	−0.75	−1.83	0.40	5.85	1.65	8.03	1.69	−1.74	1.56
-funit-at-a-time	2.48	0.74	−0.21	0.40	7.25	−1.68	10.00	2.56	−3.45	1.56
-fstrict-aliasing	0.00	0.00	−0.21	0.00	2.35	−6.56	9.00	0.84	0.90	0.77
-fomit-frame-pointer	2.48	−0.75	−0.41	0.00	4.34	−0.58	6.19	0.00	−0.88	0.77
-fgcse	0.60	0.00	0.00	−2.18	3.33	6.50	6.14	0.84	−2.61	0.76
-finline-functions	1.85	−0.75	−0.31	0.20	7.42	−9.40	2.56	0.84	−2.59	0.00
-freorder-blocks	0.00	−0.75	0.00	0.10	4.32	−5.24	6.14	−1.64	0.00	0.00
-frename-registers	0.60	−1.49	0.40	0.40	5.85	−1.66	0.82	0.84	−1.79	0.00
-foptimize sibling-calls	−0.61	0.00	−1.42	0.10	2.35	−4.66	5.21	0.00	−1.76	0.00
-fif-conversion	0.00	0.00	0.20	0.20	0.94	1.10	4.31	−0.84	−0.90	0.00
-funroll-loops	0.60	−2.99	−1.01	0.10	1.87	−3.98	0.83	0.00	−0.88	−0.77
-fold-unroll-loops	6.66	−0.75	0.20	2.43	−36.75	3.48	−4.96	1.66	3.63	−3.08
-ffast-math	−0.60	0.00	0.10	0.30	−0.47	2.90	−5.47	−0.83	−2.59	−0.76
-fpic	0.63	0.00	−0.21	0.20	−2.04	2.95	0.00	0.00	0.87	0.00
-funroll-all-loops	0.00	−0.75	0.71	0.70	0.00	7.55	−0.82	−4.17	−1.79	0.00
-fpeel-loops	3.63	0.00	0.20	6.06	0.00	4.06	−4.14	0.00	0.00	0.76

File size (relative increase of the size of stripped binaries in percents):

options	wupwise	swim	mgrid	applu	mesa	art	equake	ammp	apsi	total
-fbranch-probabilities	0.37	−0.11	0.20	0.15	−7.43	−6.42	−0.06	−0.92	−2.47	−4.77
-funit-at-a-time	0.37	−0.11	0.20	0.15	−7.37	−6.42	0.57	0.03	−2.47	−4.61
-fomit-frame-pointer	0.00	−0.53	−1.53	−0.35	−3.45	−7.19	−2.12	−4.38	−1.30	−2.96
-fgcse	0.00	−26.92	0.57	−8.87	−1.06	−7.19	0.25	−0.02	−0.74	−1.93
-fstrict-aliasing	0.00	0.00	0.00	0.00	−0.31	−7.19	−2.17	−0.10	−0.15	−0.44
-fif-conversion	0.00	−0.21	−0.09	−0.08	−0.22	−0.73	0.05	0.31	−0.03	−0.11
-foptimize sibling-calls	0.00	0.00	0.00	0.00	−0.04	0.00	−0.06	−0.01	0.00	−0.02
-freorder-blocks	0.00	0.10	0.00	0.02	0.28	−0.19	0.11	−0.43	−0.20	0.07
-funroll-loops	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.02	0.00
-frename-registers	0.00	0.10	0.16	−0.08	0.13	0.18	0.22	0.11	−0.32	0.05
-finline-functions	0.37	0.20	0.28	0.13	−0.74	19.29	8.46	1.21	−1.87	0.09
-ftracer	8.22	0.00	0.08	0.02	0.22	0.55	0.22	1.01	1.72	0.79
-fbranch-probabilities -fguess-branch...	9.36	0.84	1.36	−1.20	0.00	2.79	0.97	4.33	1.80	1.17
-fstrength-reduce	7.50	2.68	3.28	7.17	−0.17	0.00	0.22	0.37	7.24	1.70
-fschedule-insns2	3.87	2.47	3.73	6.90	5.47	3.11	4.91	5.97	6.53	5.64
-fschedule-insns -fschedule-insns2	3.78	2.47	4.26	11.04	5.55	3.11	5.66	5.97	6.97	6.25
-fpic	−1.98	−0.42	0.24	−2.57	−0.09	2.76	1.10	−1.06	−0.08	−3.83
-ffast-math	−0.21	−2.00	−1.05	−0.97	0.27	0.30	−0.60	−0.76	−0.61	−0.17
-fpeel-loops	0.00	0.00	0.00	0.90	0.00	7.73	2.60	0.00	0.00	0.30
-funroll-all-loops	0.00	0.00	0.81	0.29	0.00	7.73	1.13	0.37	0.27	0.34
-fold-unroll-loops	2.71	36.40	15.56	5.15	4.52	23.73	6.75	15.75	8.91	7.79