Evaluation of virtual machine performance

Rafael Timbó Matos * Edson Borin †

1 Introduction

This report presents the results achieved and the methodology used on the performance analysis of some virtual machines. We first performed a survey of benchmarks and selected the SPEC CPU 2006 benchmarks to experiment with QEMU. The SPEC CPU is composed of 29 C, C++ and Fortran programs and are typically used for processor performance analysis. After selecting these benchmarks, we generated x86-32, x86-64, and ARM binaries for these benchmarks. Next, we installed the QEMU, DynamoRIO and Pin virtual machines and executed the generated binaries natively and using the virtual machines on x86 and QEMU on ARM systems. Our results indicate that QEMU is really slow, while DynamoRIO and Pin are at most 10 times slower when compared to native execution.

2 Materials and Methodology

2.1 Materials

This section describes the environment setup. The ARM platform used was an i.MX53 Quick Start Board based on the Freescale Semiconductor i.MX53 Applications Processor [**?**]. Its Cortex-A8 processor with 1GHz of frequency is built around a system with hardware and multimedia accelerators, graphics processing units, which have support to 2D and 3D graphics. Moreover, the platform has 1GB of RAM memory, ethernet connection, USB, PATA, SATA, and VGA. It also comes with a 4GB SD card preloaded with Ubuntu Demonstration Software.

Due to the chip's low memory, we needed a swap memory, for which the SD card is not appropriate. Therefore we set up a 120GB hard disk, copying the operational system to it with the Unix `dd` command. Then a 4GB swap memory was set up using the Gparted partition editor. Next, we had to reconfigure the board boot loader, U-boot [**?**], to load the operational system from the disk rather than from the SD card. After properly configuring the hard disk, the swap memory and the boot loader, the ARM system was ready to run the SPEC tests. SPEC CPU is a software benchmark set used as a measure to compare performance of different computer systems. It is based on real application software. To ease

*Institute of Computing, UNICAMP, Brazil. Research conducted with partial financial support from CNPq.

†Institute of Computing, UNICAMP, Brazil.

SPEC use, some tools are provided with the SPEC CPU benchmarks. These tools aid users to collect result data and run tests. However, their precompiled binaries are unavailable for ARM architectures. The tools were compiled for our ARM system with the `buildtools` script provided by SPEC.

Afterwards, the benchmarks themselves needed to be compiled. Checking our libraries and programs were necessary, for some were missing, such as gfortran, the GNU compiler for the Fortran language, gmp, the GNU multiple-precision arithmetic library, mpfr, the GNU multiple-precision floating-point computations. The flags employed were `-O3`, which instructs the compiler to perform code optimizations, and `-lm`, responsible for linking against the math library. The runspec option "`--build`" were used in order to separate the compilation phase from the running phase. The metric utilized for compilation was the base, defined by SPEC. The base guideline states that the compilation must be made using a single set of switches in a single-step make process. This means that all benchmarks of a given language must be compiled using the same flags in the same order and no feedback-directed optimization is allowed.

As for the x86 platform, the machine used was one from the Unicamp's Computer Systems Laboratory (LSC) with an 2.4GHz Intel Xeon CPU E5645 processor assembled with 32GB of memory and loaded with Ubuntu Server 10.04.3 LTS amd64. The support tools were already precompiled in this case and the `linux-suse101-i386` tools were employed. The compilation process for the x86 environment is basically the same as the one described above for the ARM system. Variables used on configuration files are given in section 2.2.

Finally, we needed to set up our virtual machines, user-mode QEMU, DynamoRIO, and Pin, detailed in the subsections below.

2.1.1 QEMU

The QEMU tool [**?**, **?**] is a virtual machine that relies on dynamic binary translation to emulate x86 (32 bits), x86-64, ARM, MIPS, PowerPC, Sparc32, Sparc64 and a few other CPU binaries. QEMU version 1.1.1 was compiled on both systems using the commands presented on table 1.

The dynamic translation is done splitting target instructions into simpler instructions called *micro operations*. Then a compile time tool - called *dyngen* - uses these operations to create a dynamic code generator, used at runtime to generate a complete host function containing several micro operations. These are used to create basic blocks of host code called *Translated Blocks*. Translated blocks are considered to be associated with a immutable CPU internal state in order to improve QEMU's performance. Once CPU state information changes, another translated block will be generated. As a result, the state can be recorded with the translated block itself. For example, jumping, branching or changing privilege level are actions that can change a CPU state in a unknown way at translation time. Translated Blocks are stored in a 32MB code cache, being completely flushed when full for simplicity. Moreover, when translated code is generated for a basic block, the corresponding host page is write protected if it is not already read-only. Then, if a write access is done to the page, Linux raises a SEGV signal. QEMU then invalidates all the translated code in the page and enables write accesses to the page. Correct translated code invalidation is done

efficiently by maintaining a linked list of every translated block contained in a given page. Other linked lists are also maintained to undo direct block chaining. In x86 emulation, no instruction cache invalidation is signaled by the application when code is modified, turning some intrinsic topics, such as self-modifying code, into a special challenge.

To improve performance, QEMU keeps some of its features simple. For instance, QEMU uses a fixed register allocation approach. That is, each target CPU register is mapped to a fixed memory address or host register. Typically, the former holds the target registers and the latter is used for temporary variables. Concerning hardware interrupts, translated blocks do not check for them. This is done only by the main loop. However, interrupt checking can be manually triggered by asynchronously calling a specific function to reset the chaining of the currently executing translated block. This ensures that the main loop will soon regain the execution control, therefore handling pending interrupts. As for exceptions, the C function *longjmp* is used. The host SIGSEGV and SIGBUS signal handlers are used to get invalid memory accesses. The simulated program counter is found by retranslating the corresponding basic block and by looking where the host program counter was at the exception point. The emulated code can be restarted in case of need. Another example of simplified behavior is the processing of condition codes. Some processors are built to set the condition codes automatically on each instruction, even though whether these codes will be used is unknown. This can be a major source of overhead and, to overcome this, QEMU uses lazy evaluation of CPU condition codes. In other words, the tool only calculates the condition codes when needed. It does so by storing just one operand, the result, and the type of operation. The block is then further optimized at translation time by removing unused values.

QEMU also enhances block chaining. Once the execution of a translated block has finished, the simulated Program Counter and the CPU state are used as a hash to find the next block. If the translation has already been done, a jump to the block is done. Otherwise, a new translation is initiated. An improvement is done by patching a translated block so that it jumps directly to the next one, therefore reducing the runtime overhead. This accelerates the most common case where the new simulated PC is known.

On the subject of memory management, QEMU uses the *mmap* system call to emulate the target MMU, which works as long as the emulated operational system does not use reserved areas by the host operational system. QEMU also supports a software MMU, in which the MMU virtual to physical address translation is done at every memory access. An address translation cache is used to speed up the translation. Moreover, Each translated block is indexed with its physical address. When MMU mappings change, the chaining of the translated blocks is reset, because the jump target may change.

Beyond emulating a whole operating system, QEMU is also capable of emulating a Linux process, with special treatment for system calls, signals and threads. A generic system call translator for Linux is responsible for solving endianness and 32/64 bits issues. Besides, the ioctl system call is specially handled with a generic type description system. It also emulates the *mmap* system call to support host CPU pages bigger than 4KB. Similarly to the Linux kernel, normal and real-time signals are queued along with their information and, when the virtual CPU is interrupted, one queued signal is handled by generating a stack frame in the virtual CPU. Then, a *sigreturn* system call is emulated to return from the virtual signal

handler. When possible, the signal handling is done directly by the host Linux Kernel. For instance, SIGALRM and the blocked signal mask are not emulated, allowing most signal system calls to be redirected directly to the host Linux Kernel. On the other hand, CPU exceptions, such as SIGFPE, and the *sigaction* system call are emulated. In order to emulate threads, QEMU uses the host *clone* system call to create real host threads, creating one virtual CPU instance for each thread. For the x86 architecture, atomic operations are emulated with a global lock to preserve their semantic.

2.1.2 DynamoRIO

DynamoRIO [**?**] is runtime code manipulation system that supports code transformations on any part of a program, while it executes. On our work, we only used it as a process virtual machine and did not perform any code manipulations. The binaries used were downloaded directly from its site [**?**]. DynamoRIO operates in user mode on a target process. As a process virtual machine, it interposes between the application and the operating system. It has a complete view of the application code stream and acts as a runtime control point. The application itself, along with the underlying operating system and hardware, remain unchanged.

DynamoRIO operates by shifting an application's execution from its original instructions to a code cache, which greatly improves execution in comparison to emulation. DynamoRIO occupies the address space with the application and has full control over execution, taking over whenever control leaves the code cache or when the operating system directly transfers control to the application. This is achieved by modifying control transfer instructions, such as branches and jumps, to give control to DynamoRIO.

Furthermore, the application code is copied one dynamic basic block at a time into DynamoRIO's basic block code cache. Address conversion between the cache and the application are needed for indirect branch address translation and handling of signals and exceptions, for example. Indirect branches require dynamic resolution of their targets, which is performed via an inlined table lookup or a comparison to a known target inlined into a trace. A block that directly targets another block already resident in the cache is linked to that target block to avoid the cost of returning to the DynamoRIO dispatcher, which considerably affects performance, as returning to the dispatcher would require context switching. Frequently executed sequences of basic blocks are combined into traces, placed in a separate code cache. Despite duplicated codes, these linking techniques are generally an improvement, as they achieve better code layout for a significant performance boost and create less data structures related to basic blocks.

DynamoRIO executes applications transparently, leaving program binaries and data unmodified, but sacrificing some aggressive optimizations that would assume a behavior from the application. Moreover, DynamoRIO relies only on system calls and never on user libraries for external resources due to reentrancy and corruption problems. DynamoRIO also maintains errors transparency: applications that have faulty behavior, such as invalid memory access or invalid instruction execution, are handled as if the fault occurred natively. Nevertheless, debugger threads that should run natively currently does not, although debuggers such as gdb or Debugging Tools for Windows work fine with DynamoRIO.

Additionally, DynamoRIO maintains its own stack, I/O routines and memory manager, obtaining its memory directly from system calls and separated from the application's memory. This prevents resource usage conflicts and helps maintaining the application unaware of DynamoRIO.

2.1.3 Pin

Similarly to DynamoRIO, Pin [**?**] consists of a virtual machine, a code cache and an instrumentation API which enable users to manipulate application code. Its binaries were downloaded already compiled from its site [**?**]. Pin virtual machine consists of a just-in-time compiler (JIT), an emulator, and a dispatcher. The application is emulated based on the following algorithm: The JIT compiles and instruments application code, which is then launched by the dispatcher and stored in the code cache, the only area where application code is executed. Switching context between the virtual machine and the code cache involves saving and restoring the application register state. The emulator is responsible for the interpretation of instructions that cannot be executed directly and for system calls that require special handling from the virtual machine.

Pin is injected into the address space of an application. Unlike DynamoRIO, Pin uses the Unix Ptrace API to gain control. After initializing, Pin loads the Pintool, that is, the tool created using Pin instrumentation API, and starts it running. Then the Pintool initializes itself and requests application start-up. Pin creates the initial context and starts jitting the application. Ptrace mechanism allows Pin to attach to an already running process and to detach from an instrumented process, which then executes the original, uninstrumented code. Although applications share the same address space as Pin, they do not share any libraries, possibly resulting in more than one copy of libraries. This avoids unwanted interactions between them, as the application will have its own copy of a non-reentrant library.

Pin injection method has three advantages over the technique used by DynamoRIO. First, it works with statically-linked binaries, which is the case of this experiment. However, this DynamoRIO flaw can be overcome by explicitly calling `drloader` to inject DynamoRIO, further detailed in section 2.2. Second, loading an extra shared library will shift all of the application shared libraries and some dynamically allocated memory to a higher address when compared to an uninstrumented execution. This further and undesirably modifies the original behavior of the application. Third, the instrumentation tool cannot obtain control of the application until after the shared-library loader has partially executed, while Pin's method is able to instrument the very first instruction in the program.

Furthermore, Pin JIT compiles directly into the same ISA without relying on intermediate formats. An application is compiled one trace at a time. The Pin concept of trace is a straight-line sequence of instructions which terminates at one of the conditions: (i) an unconditional control transfer (branch, call, or return), (ii) a predefined number of conditional control transfers, or (iii) a predefined number of instructions have been fetched in the trace. This implies a trace can have multiple side-exits in addition to the last exit. An exit initially branches to a stub, which redirects the control to the virtual machine. The resolution of the target address then takes place, generating a new trace for the target if it

has not been generated before, and execution resumes at the target trace.

Given this Pin basic functionality, it has additional features which attempt to improve its performance. For instance, Pin tries to bypass stubs and the virtual machine by branching a trace exit directly to its target trace, a process called trace linking. Direct jump resolution is simply patching the branch at the end of one trace to jump to the target trace. An indirect control transfer, however, has multiple possible targets and therefore needs some sort of target-prediction mechanism. Pin's approach translates the indirect jump into a move and a direct jump, which is the first predicted address. In case of wrong prediction, another prediction target is selected. Another failure will cause Pin to branch to a hash table in search for the target. As a last resort, the control can be transfered to the virtual machine to indirect target resolution.

Pin's indirect linking mechanism holds three main differences compared to the approach taken by DynamoRIO. First, The entire chain of DynamoRIO is generated at one time and embedded at the translation of the indirect jump, preventing any later predicted target to be added onto the chain. In contrast, Pin mechanism builds the chain incrementally, allowing newly seen targets to be inserted in any order. The second difference is that Pin hash table is local, whereas DynamoRIO uses a global hash table for all indirect jumps. Local hash table has been shown to typically offer higher performance [**?**]. The third difference is that Pin applies function cloning to accelerate returns, the most common form of indirect control transfers.

Another Pin feature aiming performance is its register reallocation mechanism. The JIT must ensure that inserted calls does not overwrite any application registers and that these virtual registers matches its physical bindings at the entrance of a destination trace, which adds some emulation overhead. Pin minimizes this using register liveness analysis and reconciliation of register bindings, based on traces. A dead register, for example, can be reused without spilling it to memory. Pin keeps a virtual register in the same physical register across traces whenever possible, and generates compensation code to reconcile register bindings when they differ. This approach is more efficient than flushing registers to memory, as bindings show differences in only one or two virtual registers in practice [**?**]. As a consequence, Pin must remember the binding at a trace's entry. The definition of Pin trace can therefore be resumed to the pair of its entry address and some static properties that hold at the trace's entry, called static context. The JIT will only generate a new trace if it cannot find a compatible one. Two traces are compatible if they have the same address and their static context are either identical or different in only their register bindings, which can be overcome by reconciling the bindings.

As most of Pin instrumentation slowdown comes from executing the instrumentation code rather than the compilation time [**?**], Pin spend some time analyzing and optimizing instrumentation code. For example, frequently-executed counting and tracing are inlined. Without inlining, Pin would call a bridge routine that would save all caller-saved registers, set up analysis routine arguments, and finally call the analysis routine. In other words, inlining eliminates the bridge, saving two calls and two returns, and exchanges the expensive caller-saved register saving for a register renaming and, if needed, spilling. Furthermore, inlining enables other optimizations like constant folding of analysis routine arguments.

Moreover, an additional optimization is performed exclusively for the x86 eflags, as most

analysis routines modify this conditional flags register. Pin uses liveness analysis on eflags, frequently discovering eflags are dead, therefore eliminating savings and restores. This optimization is important due to expensive stacks operations needed to access this register. Finally, the Pintool writer can specify when a analysis routine can be inserted anywhere inside a basic block or a trace, opening a number of optimization opportunities through call scheduling.

2.2 Methodology

This section depicts the construction of configuration files used, how the virtual machines were compiled or downloaded, and some assumptions made in order to make a better display of results.

SPEC benchmarks are subdivided in two sets: specint, the INT set, composed mainly of integer operations, whose benchmarks are 410.bwaves, 416.gamess, 433.milc, 434.zeusmp, 435.gromacs, 436.cactusADM, 437.leslie3d, 444.namd, 447.dealII, 450.soplex, 453.povray, 454.calculix, 459.GemsFDTD, 465.tonto, 470.lbm, 481.wrf, and 482.sphinx3; and specfp, the FP set, which focus on floating-point operations, whose benchmarks are 400.perlbench, 401.bzip2, 403.gcc, 429.mcf, 445.gobmk, 456.hmmer, 458.sjeng, 462.libquantum, 464.h264ref, 471.omnetpp, 473.astar, and 483.xalancbmk. Taking that on board, any citation of INT or FP benchmarks refers to these set of benchmarks, unless stated otherwise.

With the SPEC tools already compiled, the SPEC runs could be initiated. All runs were made through the invocation of the `runspec` tool, which called requested benchmarks, saving any errors and run statistics, such as running times. In order to reduce random noises on the experiments, some CPU-consuming services were also disabled, such as gdm - GNOME display manager -, cups - the Common Unix Printing System -, pulseaudio - a sound server -, and the ssh server, which enables remote access.

Tables 2 and 3 further detail configurations about the experiments. These settings can be used on SPEC configuration files, attributing the values on the right to the variables on the left by the use of an equal sign. The `rebuild` flag were only set to 0 after the benchmarks were already compiled, thus avoiding accidental rebuilding. Note that SPEC tools include hashes on the configuration files used to compile the benchmarks. Taking that on board, we need to copy the hashes to other configuration files if we do not want the benchmarks to be recompiled. On our experiment, we generated the configuration files as needed, based on existent ones. Thus, in the end, all configuration files responsible for emulation were generated from the configuration file that had made the compilation, therefore containing the hashes and skipping compilation on future runs.

Plus, the emulation variables were only set when the benchmarks were being emulated, according to the virtual machine that was doing the emulation. Each line of submit was used if and only if the virtual machine in parenthesis were used. If there is no parenthesis, that line was always used. Thus, in QEMU-emulated runs, runspec's `submit` variable would evaluate to "`${QEMUBIN} $command`", where `QEMUBIN` is the variable that holds the full path to the QEMU binary, set in the configuration file. On our experiment, we generated the configuration file for QEMU first, then generated DynamoRIO and Pin configurations based on the previously generated QEMU file.

Qemu Compilation Commands

```
mkdir build
mkdir install
cd build
unset LD_CONFIG_PATH
unset PKG_CONFIG_PATH
export LD_CONFIG_PATH=/path/to/glib/lib
export PKG_CONFIG_PATH=/path/to/glib/lib/pkgconfig
../qemu-1.1.1/configure --disable-system --enable-user \
--disable-sdl --disable-virtfs --disable-vnc --disable-brlapi \
--disable-bluez --disable-slirp --disable-kvm --disable-vhost-net \
--disable-libiscsi --disable-smartcard --disable-smartcard-nss \
--disable-usb-redir --disable-sparse --disable-debug-tcg \
--disable-xen --disable-vnc-tls --disable-vnc-sasl \
--disable-vnc-jpeg --disable-vnc-png --disable-vnc-thread \
--disable-curses --disable-curl \
--target-list=i386-softmmu,x86_64-softmmu,arm-softmmu, \
i386-linux-user,x86_64-linux-user,arm-linux-user,armeb-linux-user \
--block-drv-whitelist=oss,alsa,sdl,esd,pa,fmod --static \
--extra-cflags="-O3 -m64 -static -L/path/to/glib/lib" \
--extra-ldflags=-L/path/to/glib/lib \
--prefix=../install \
&& make LDFLAGS="-L/path/to/glib/lib"
```

Table 1: QEMU Compilation Commands

The QEMU softwares used in our experiments were `qemu-arm` for ARM, `qemu-i386` for 32-bits x86, and `qemu-x86_64` for 64-bits x86 emulation. Other binaries generated in the compilation process for QEMU were not used. QEMU version 1.1.1 was compiled on both systems using the commands presented on table 1. However, QEMU required a newer glib version than the one we had in our systems, so we needed to compile it for ARM and 32-bits and 64-bits x86 as well, which was done using its source code [**?**] only explicitly setting the `-O3` and the word size flags. According to our results, presented on section 3, QEMU achieved a performance of a hundred times slower on some benchmarks. This overhead is unnecessarily slow to the emulation standards [**?**]. We decided then to investigate the causes of this overhead. The perf [**?**] tool version 2.6.32-46 was used to profile the SPEC emulation done by QEMU. The `runspec` variable on the configuration file was changed to invoke perf as well. Specifically, in these profiling runs, runspec's `submit` variable would evaluate to "`${PERFBIN} record -A -o ${PERFOUT} ${QEMUBIN} $command`", where `PERFBIN` is the perf executable, `PERFOUT` is the file perf will place data, and `QEMUBIN` is the QEMU binary.

Profiling runs were done only in 64-bits qemu-x86_64, as they were enough to reveal QEMU slowness causes. Profiling run results consist basically of time spent on each area of code. Perf, more specifically, uses counters to statistically determine how much time was

spent on each symbol. We gathered these times in meaningful pieces to our experiment according to the following criteria:

Benchmark Time spent in the benchmark itself. A benchmark code can be identified in our profiling results as rows that have `qemu-x86_64` as a command and a hexadecimal address as its symbol. The explanation behind this association is that the benchmark is fetched from its file into the dynamically-allocated code cache of QEMU. As emulators can not afford to manage symbols for performance reasons, no information becomes related to their code, except those needed, such as hexadecimal addresses, used by jumps. This reasoning is sustained because QEMU does not use dynamically-generated code other than the benchmark code itself. Also, we rebuilt QEMU for profiling runs using the same configuration as before, but no `make install` command was issued, as it would strip - or discard the symbols of - QEMU binaries. Instead, we collected its binaries manually from the build directory.

Mutex Time spent in the `__pthread_mutex_unlock` and the `__pthread_mutex_lock` functions. These functions are responsible for synchronizing processes or threads, ensuring no two of them are concurrently accessing a shared resource.

SoftFloat Time spent in functions whose implementation was in the source-code *fpu* directory. All files in that directory, as in qemu version 1.1.1, have their names starting with "softfloat", which is the IEC/IEEE Floating-point Arithmetic Package that emulates floating-point operations with software.

Qemu Time spent in qemu functions other than the ones already specified above. This area is the emulation of code discounting the time spent on mutexes and floating-point software emulation.

Perf Time spent in the profiling tool itself. This area does not affect our results as it does not reach a high percentage and discrepancy of times in other areas still show.

The areas does not always add up to exactly 100% due to rounding, but they always reach at least 99.5%.

Then, we moved to DynamoRIO. DynamoRIO was obtained directly from its site [**?**]. We did not compile it ourselves. We used the version 4.0.1-1. Its configuration file is really similar to the QEMU one. Therefore, we used a script to generate the DynamoRIO configuration file based on QEMU configuration, presented on table 4. The tweaks made were the necessary ones to correctly emulate the benchmarks with DynamoRIO.

For example, our benchmarks were compiled statically and, as such, they do not invoke the linker in order to run. This prevents the injection of DynamoRIO code on the application, as the default mode of DynamoRIO insertion is via the environment variable `LD_PRELOAD`, which is handled by the linker. Given that our benchmarks were statically compiled, we needed an alternative method of injection. Skimming the `drrun` script, we could see that the undocumented option `-early` triggers a `drloader` call with the DynamoRIO main library as its first argument and the application as following arguments. This early injects DynamoRIO and fakes the application as an argv[0], sufficing our needs. Optionally, `-32` and `-64` options can be added to the submit parameters when the word size of DynamoRIO and of the benchmarks differs.

Next, we ran the Intel Pin framework. Pin binaries was downloaded from Intel site [**?**]. The version used on this experiment was 2.12-58423, compiled with gcc 4.4.7. As before, the

64-bits x86 Configuration	
tune	base
runlist	all
rebuild	0
Compilers	
CC	gcc 4.6.3
CXX	g++ 4.6.3
FC	gfortran 4.6.3
Emulation Variables	
`submit1 (perf)`	`${PERFBIN} record`
`submit2 (perf)`	`-A -o ${PERFOUT}`
`submit3 (dynamorio)`	`${DYNBIN} [-64]`
`submit4 (pin)`	`${PINBIN} --`
`submit5 (QEMU)`	`${QEMUBIN}`
`submit6`	`$command`
`use_submit_for_speed`	yes
Optimization flags	
COPTIMIZE	-O3
CXXOPTIMIZE	-O3
FOPTIMIZE	-O3
EXTRA_OPTIMIZE	-static
Portability Flags	
default:	
PORTABILITY	-DSPEC_CPU_LP64
400.perlbench:	
CPORTABILITY	-DSPEC_CPU_LINUX_X64
EXTRA_LDFLAGS	-lm
403.gcc:	
EXTRA_LDFLAGS	-lm
433.milc:	
EXTRA_LDFLAGS	-lm
445.gobmk:	
EXTRA_LDFLAGS	-lm
456.hmmer:	
EXTRA_LDFLAGS	-lm
462.libquantum:	
CPORTABILITY	-DSPEC_CPU_LINUX
EXTRA_LDFLAGS	-lm
464.h264ref:	
EXTRA_LDFLAGS	-lm
470.lbm:	
EXTRA_LDFLAGS	-lm
482.sphinx3:	
EXTRA_LDFLAGS	-lm
483.xalancbmk:	
CXXPORTABILITY	-DSPEC_CPU_LINUX
481.wrf:	
CPORTABILITY	-DSPEC_CPU_CASE_FLAG -DSPEC_CPU_LINUX

32-bits x86 Configuration	
tune	base
runlist	all
rebuild	0
Compilers	
CC	gcc 4.6.3
CXX	g++ 4.6.3
FC	gfortran 4.6.3
Emulation Variables	
`submit1 (perf)`	`${PERFBIN} record`
`submit2 (perf)`	`-A -o ${PERFOUT}`
`submit3 (dynamorio)`	`${DYNBIN} [-32]`
`submit4 (pin)`	`${PINBIN} --`
`submit5 (QEMU)`	`${QEMUBIN}`
`submit6`	`$command`
`use_submit_for_speed`	yes
Optimization flags	
COPTIMIZE	-O3
CXXOPTIMIZE	-O3
FOPTIMIZE	-O3
EXTRA_OPTIMIZE	-m32 -static
Portability Flags	
default:	
PORTABILITY	-DSPEC_CPU_ILP32
400.perlbench:	
CPORTABILITY	-DSPEC_CPU_LINUX_IA32
EXTRA_LDFLAGS	-lm
403.gcc:	
EXTRA_LDFLAGS	-lm
433.milc:	
EXTRA_LDFLAGS	-lm
445.gobmk:	
EXTRA_LDFLAGS	-lm
456.hmmer:	
EXTRA_LDFLAGS	-lm
462.libquantum:	
CPORTABILITY	-DSPEC_CPU_LINUX
EXTRA_LDFLAGS	-lm
464.h264ref:	
EXTRA_LDFLAGS	-lm
470.lbm:	
EXTRA_LDFLAGS	-lm
482.sphinx3:	
EXTRA_LDFLAGS	-lm
483.xalancbmk:	
CXXPORTABILITY	-DSPEC_CPU_LINUX
481.wrf:	
CPORTABILITY	-DSPEC_CPU_CASE_FLAG -DSPEC_CPU_LINUX

Table 2: x86 Configuration

ARM Configuration	
tune	base
runlist	all ^464.h264ref ^482.sphinx3
rebuild	0
Compiler	
CC	gcc 4.4.3
CXX	g++ 4.4.3
FC	gfortran 4.4.3
Emulation Variables	
`submit1 (QEMU)`	`${QEMUBIN}`
`submit2`	`$command`
`use_submit_for_speed`	yes
Optimization flags	
COPTIMIZE	-O3
CXXOPTIMIZE	-O3
FOPTIMIZE	-O3
EXTRA_OPTIMIZE	-static
Portability Flags	
default:	
PORTABILITY	-DSPEC_CPU_ILP32
400.perlbench:	
CPORTABILITY	-DSPEC_CPU_LINUX
EXTRA_LDFLAGS	-lm
403.gcc:	
EXTRA_LDFLAGS	-lm
433.milc:	
EXTRA_LDFLAGS	-lm
445.gobmk:	
EXTRA_LDFLAGS	-lm
456.hmmer:	
EXTRA_LDFLAGS	-lm
435.gromacs:	
COPTIMIZE	-O1
462.libquantum:	
CPORTABILITY	-DSPEC_CPU_LINUX
EXTRA_LDFLAGS	-lm
483.xalancbmk:	
CXXPORTABILITY	-DSPEC_CPU_LINUX
481.wrf:	
CPORTABILITY	-DSPEC_CPU_CASE_FLAG -DSPEC_CPU_LINUX

Table 3: ARM Configuration

Generating DynamoRIO config from QEMU's	
Environment Variables	
`SRC_FILE`	QEMU configuration file
`NEW_FILE`	DynamoRIO file to be generated
Commands	

`cp $SRC_FILE $NEW_FILE`
`sed -i "s#QEMUBIN=.*#DYNAMORIOBIN=$EXE#" $NEW_FILE`
`sed -i "s#submit=.*#submit=${DYNAMORIOBIN} $command#" $NEW_FILE`	

Table 4: DynamoRIO Configuration Setup

Generating Pin config from QEMU's	
Environment Variables	
`SRC_FILE`	QEMU configuration file
`NEW_FILE`	Pin file to be generated
Commands	

`cp $SRC_FILE $NEW_FILE`
`sed -i "s#QEMUBIN=.*#PINBIN=$EXE#" $NEW_FILE`
`sed -i "s#submit=.*#submit=${PINBIN} -- $command#" $NEW_FILE`	

Table 5: Pin Configuration Setup

configuration file was slightly changed. Table 5 presents the script to create pin configuration file based on QEMU one. Only two configuration files were created for DynamoRIO and Pin. This is sufficient as Pin or DynamoRIO invoked to emulate an application of dissimilar word size simply transfers control to the binary of appropriate word size.

3 Results

This section shows the results from SPEC CPU experiments and QEMU profiling. A zero indicates that the run experiment for that benchmark could not be made.

The base reference time shown on tables is based on a reference machine used by SPEC to normalize the performance metrics used. Each benchmark is run and measured on that reference machine to establish a reference time for that benchmark, presented on run time tables. These times are then used in the SPEC calculations as a reference ratio of 1.00. Thus, the numbers presented on ratio tables below express how many times our systems were faster, in case of values greater than one, or slower, in case of values less than one, than SPEC reference machine for SPEC CPU benchmarks.

Note that QEMU wordsize does not define the word size of the benchmark, as the word size of QEMU binaries defines where QEMU should run natively, not what QEMU will emulate. For instance, a 32-bits `qemu-i386` emulates 32-bits x86 binaries, as 32-bits `qemu-x86_64` emulates 64-bits x86 binaries, but both were generated to run on 32-bits x86 architecture. In other words, each table or graphic refers to the set of benchmarks compiled for either 32 or 64 bits, containing results of native runs, 32-bits QEMU, 64-bits QEMU, DynamoRIO and Pin emulated runs.

The charts show us that QEMU x86 emulation is on average 42 times slower than native execution for the SPEC CPU benchmark, being 32-bits binaries of QEMU slightly faster than 64-bits ones. However, while some benchmark performances were only a few times slower when emulated, others had their execution time increased by more than a hundred, revealing a high discrepancy in emulation among QEMU emulation of SPEC CPU benchmarks. As for the emulation of DynamoRIO and Pin, the results showed them to be affordable, revealing most of their emulation between one and two times slower, rarely reaching the mark of 4 times slower.

Benchmark	**Base Ref Time**	**32-bits Native**	**32-bits qemu-i386**	**64-bits qemu-i386**
INT Benchmarks				
400.perlbench	9770	411.88	5 028.70	4 688.71
401.bzip2	9650	725.75	3 270.97	3 263.77
403.gcc	8050	378.94	2 354.51	2 231.64
429.mcf	9120	291.32	821.39	803.71
445.gobmk	10490	540.30	4 511.54	4 528.63
456.hmmer	9330	636.15	4 094.95	4 058.15
458.sjeng	12100	619.58	5 237.68	5 062.33
462.libquantum	20720	613.06	2 330.49	2 343.90
464.h264ref	22130	827.16	7 744.28	7 333.76
471.omnetpp	6250	346.11	3 323.60	2 767.14
473.astar	7020	529.36	1 824.69	1 799.49
483.xalancbmk	6900	276.35	2 790.85	2 680.05
FP Benchmarks				
410.bwaves	13590	552.59	38 913.33	28 608.02
416.gamess	19580	1 129.62	82 245.39	57 239.08
433.milc	9180	526.03	33 210.60	23 688.21
434.zeusmp	9100	685.07	45 188.87	31 792.80
435.gromacs	7140	1 037.14	71 764.80	51 629.49
436.cactusADM	11950	1 389.47	106 767.04	76 641.38
437.leslie3d	9400	612.72	41 601.22	29 098.35
444.namd	8020	593.92	65 015.86	45 688.74
447.dealII	11440	426.40	18 149.37	13 143.83
450.soplex	8340	306.55	7 437.71	5 279.34
453.povray	5320	278.25	14 758.39	10 715.80
454.calculix	8250	920.36	170 509.31	117 770.74
459.GemsFDTD	10610	584.71	46 852.43	32 381.39
465.tonto	9840	719.58	35 336.10	24 904.17
470.lbm	13740	472.53	56 600.73	38 232.58
481.wrf	11170	963.49	62 300.50	46 447.25
482.sphinx3	19490	555.76	60 988.44	44 253.51

Table 6: 32-bits x86 Benchmarks Run Times for QEMU

Benchmark	**32-bits Native**	**32-bits qemu-i386**	**64-bits qemu-i386**
INT Benchmarks			
400.perlbench	23.72	1.94	2.08
401.bzip2	13.30	2.95	2.96
403.gcc	21.24	3.42	3.61
429.mcf	31.31	11.10	11.35
445.gobmk	19.42	2.33	2.32
456.hmmer	14.67	2.28	2.30
458.sjeng	19.53	2.31	2.39
462.libquantum	33.80	8.89	8.84
464.h264ref	26.75	2.86	3.02
471.omnetpp	18.06	1.88	2.26
473.astar	13.26	3.85	3.90
483.xalancbmk	24.97	2.47	2.57
FP Benchmarks			
410.bwaves	24.59	0.35	0.48
416.gamess	17.33	0.24	0.34
433.milc	17.45	0.28	0.39
434.zeusmp	13.28	0.20	0.29
435.gromacs	6.88	0.10	0.14
436.cactusADM	8.60	0.11	0.16
437.leslie3d	15.34	0.23	0.32
444.namd	13.50	0.12	0.18
447.dealII	26.83	0.63	0.87
450.soplex	27.21	1.12	1.58
453.povray	19.12	0.36	0.50
454.calculix	8.96	0.05	0.07
459.GemsFDTD	18.15	0.23	0.33
465.tonto	13.67	0.28	0.40
470.lbm	29.08	0.24	0.36
481.wrf	11.59	0.18	0.24
482.sphinx3	35.07	0.32	0.44

Table 7: 32-bits x86 Benchmarks Ratios for QEMU

Benchmark	**Base Ref Time**	**64-bits Native**	**32-bits qemu-x86_64**	**64-bits qemu-x86_64**
INT Benchmarks				
400.perlbench	9770	424.77	0.00	6 348.34
401.bzip2	9650	591.93	4 848.25	2 927.42
403.gcc	8050	384.41	3 516.67	2 386.16
429.mcf	9120	389.12	1 221.95	964.06
445.gobmk	10490	528.87	7 512.28	4 733.16
456.hmmer	9330	487.57	7 892.55	3 803.06
458.sjeng	12100	583.13	7 978.21	5 667.07
462.libquantum	20720	513.10	2 264.51	1 346.15
464.h264ref	22130	696.23	12 558.49	7 678.96
471.omnetpp	6250	388.08	3 406.83	2 645.55
473.astar	7020	485.59	2 703.44	1 806.48
483.xalancbmk	6900	294.70	3 993.37	2 805.86
FP Benchmarks				
410.bwaves	13590	749.58	44 685.60	21 131.48
416.gamess	19580	962.10	63 457.95	30 912.82
433.milc	9180	491.63	25 795.42	13 497.58
434.zeusmp	9100	613.92	27 069.72	13 406.40
435.gromacs	7140	612.28	29 988.82	22 874.08
436.cactusADM	11950	870.80	63 978.65	33 926.38
437.leslie3d	9400	515.18	26 552.68	12 796.10
444.namd	8020	502.87	49 713.21	24 089.48
447.dealII	11440	403.38	13 728.94	7 696.35
450.soplex	8340	286.73	0.00	2 812.31
453.povray	5320	243.55	11 898.85	7 056.31
454.calculix	8250	821.43	146 003.97	68 562.55
459.GemsFDTD	10610	460.87	34 976.42	16 977.22
465.tonto	9840	573.81	30 595.41	14 826.41
470.lbm	13740	410.99	39 655.00	19 001.83
481.wrf	11170	523.05	28 949.81	21 459.41
482.sphinx3	19490	715.95	58 042.77	32 747.87

Table 8: 64-bits x86 Benchmarks Run Times for QEMU

Benchmark	**64-bits Native**	**32-bits qemu-x86_64**	**64-bits qemu-x86_64**
INT Benchmarks			
400.perlbench	23.00	0.00	1.54
401.bzip2	16.30	1.99	3.30
403.gcc	20.94	2.29	3.37
429.mcf	23.44	7.46	9.46
445.gobmk	19.83	1.40	2.22
456.hmmer	19.14	1.18	2.45
458.sjeng	20.75	1.52	2.14
462.libquantum	40.38	9.15	15.39
464.h264ref	31.79	1.76	2.88
471.omnetpp	16.11	1.83	2.36
473.astar	14.46	2.60	3.89
483.xalancbmk	23.41	1.73	2.46
FP Benchmarks			
410.bwaves	18.13	0.30	0.64
416.gamess	20.35	0.31	0.63
433.milc	18.67	0.36	0.68
434.zeusmp	14.82	0.34	0.68
435.gromacs	11.66	0.24	0.31
436.cactusADM	13.72	0.19	0.35
437.leslie3d	18.25	0.35	0.73
444.namd	15.95	0.16	0.33
447.dealII	28.36	0.83	1.49
450.soplex	29.09	0.00	2.97
453.povray	21.84	0.45	0.75
454.calculix	10.04	0.06	0.12
459.GemsFDTD	23.02	0.30	0.62
465.tonto	17.15	0.32	0.66
470.lbm	33.43	0.35	0.72
481.wrf	21.36	0.39	0.52
482.sphinx3	27.22	0.34	0.60

Table 9: 64-bits x86 Benchmarks Ratios for QEMU

Benchmark	**Base Ref Time**	**32-bits Native**	**DynamoRIO**	**Pin**
INT Benchmarks				
400.perlbench	9770	411.88	666.85	877.51
401.bzip2	9650	725.75	747.72	917.44
403.gcc	8050	378.94	0.00	771.48
429.mcf	9120	291.32	293.57	303.07
445.gobmk	10490	540.30	773.90	955.84
456.hmmer	9330	636.15	644.21	851.45
458.sjeng	12100	619.58	821.44	1 107.15
462.libquantum	20720	613.06	603.95	697.37
464.h264ref	22130	827.16	1 081.50	3 015.15
471.omnetpp	6250	346.11	404.54	462.67
473.astar	7020	529.36	531.03	615.14
483.xalancbmk	6900	276.35	349.24	469.40
FP Benchmarks				
410.bwaves	13590	552.59	559.04	599.28
416.gamess	19580	1 129.62	1 152.85	1 367.60
433.milc	9180	526.03	533.33	556.77
434.zeusmp	9100	685.07	680.71	746.79
435.gromacs	7140	1 037.14	1 042.93	1 067.24
436.cactusADM	11950	1 389.47	1 500.89	1 443.31
437.leslie3d	9400	612.72	616.81	648.91
444.namd	8020	593.92	598.52	635.59
447.dealII	11440	426.40	437.29	632.74
450.soplex	8340	306.55	317.09	371.92
453.povray	5320	278.25	321.03	433.44
454.calculix	8250	920.36	929.79	979.50
459.GemsFDTD	10610	584.71	591.51	618.49
465.tonto	9840	719.58	769.85	888.03
470.lbm	13740	472.53	473.88	474.10
481.wrf	11170	963.49	1 007.85	1 096.99
482.sphinx3	19490	555.76	579.54	726.26

Table 10: 32-bits x86 Benchmark Run Times for DynamoRIO and Pin

Benchmark	**32-bits Native**	**DynamoRIO**	**Pin**
INT Benchmarks			
400.perlbench	23.72	14.65	11.13
401.bzip2	13.30	12.91	10.52
403.gcc	21.24	0.00	10.43
429.mcf	31.31	31.07	30.09
445.gobmk	19.42	13.55	10.97
456.hmmer	14.67	14.48	10.96
458.sjeng	19.53	14.73	10.93
462.libquantum	33.80	34.31	29.71
464.h264ref	26.75	20.46	7.34
471.omnetpp	18.06	15.45	13.51
473.astar	13.26	13.22	11.41
483.xalancbmk	24.97	19.76	14.70
FP Benchmarks			
410.bwaves	24.59	24.31	22.68
416.gamess	17.33	16.98	14.32
433.milc	17.45	17.21	16.49
434.zeusmp	13.28	13.37	12.19
435.gromacs	6.88	6.85	6.69
436.cactusADM	8.60	7.96	8.28
437.leslie3d	15.34	15.24	14.49
444.namd	13.50	13.40	12.62
447.dealII	26.83	26.16	18.08
450.soplex	27.21	26.30	22.42
453.povray	19.12	16.57	12.27
454.calculix	8.96	8.87	8.42
459.GemsFDTD	18.15	17.94	17.15
465.tonto	13.67	12.78	11.08
470.lbm	29.08	28.99	28.98
481.wrf	11.59	11.08	10.18
482.sphinx3	35.07	33.63	26.84

Table 11: 32-bits x86 Benchmark Ratios for DynamoRIO and Pin

Benchmark	**Base Ref Time**	**64-bits Native**	**DynamoRIO**	**Pin**
INT Benchmarks				
400.perlbench	9770	424.77	754.21	867.56
401.bzip2	9650	591.93	624.07	684.04
403.gcc	8050	384.41	539.45	632.54
429.mcf	9120	389.12	389.18	402.88
445.gobmk	10490	528.87	815.48	903.10
456.hmmer	9330	487.57	493.24	518.17
458.sjeng	12100	583.13	852.17	1 008.81
462.libquantum	20720	513.10	502.09	548.24
464.h264ref	22130	696.23	973.55	2 796.95
471.omnetpp	6250	388.08	517.21	510.83
473.astar	7020	485.59	510.31	559.58
483.xalancbmk	6900	294.70	439.55	470.64
FP Benchmarks				
410.bwaves	13590	749.58	754.50	782.54
416.gamess	19580	962.10	1 045.48	1 136.96
433.milc	9180	491.63	508.27	512.71
434.zeusmp	9100	613.92	611.15	627.27
435.gromacs	7140	612.28	616.38	616.00
436.cactusADM	11950	870.80	890.10	865.83
437.leslie3d	9400	515.18	427.95	440.04
444.namd	8020	502.87	500.70	532.75
447.dealII	11440	403.38	466.22	536.96
450.soplex	8340	286.73	309.00	319.27
453.povray	5320	243.55	354.55	384.08
454.calculix	8250	821.43	837.68	881.31
459.GemsFDTD	10610	460.87	479.27	493.18
465.tonto	9840	573.81	666.22	738.26
470.lbm	13740	410.99	414.00	412.12
481.wrf	11170	523.05	571.86	613.22
482.sphinx3	19490	715.95	733.88	794.22

Table 12: 64-bits x86 Benchmark Run Times for DynamoRIO and Pin

Benchmark	**64-bits Native**	**DynamoRIO**	**Pin**
INT Benchmarks			
400.perlbench	23.00	12.95	11.26
401.bzip2	16.30	15.46	14.11
403.gcc	20.94	14.92	12.73
429.mcf	23.44	23.43	22.64
445.gobmk	19.83	12.86	11.62
456.hmmer	19.14	18.92	18.01
458.sjeng	20.75	14.20	11.99
462.libquantum	40.38	41.27	37.79
464.h264ref	31.79	22.73	7.91
471.omnetpp	16.11	12.08	12.24
473.astar	14.46	13.76	12.55
483.xalancbmk	23.41	15.70	14.66
FP Benchmarks			
410.bwaves	18.13	18.01	17.37
416.gamess	20.35	18.73	17.22
433.milc	18.67	18.06	17.91
434.zeusmp	14.82	14.89	14.51
435.gromacs	11.66	11.58	11.59
436.cactusADM	13.72	13.43	13.80
437.leslie3d	18.25	21.97	21.36
444.namd	15.95	16.02	15.05
447.dealII	28.36	24.54	21.31
450.soplex	29.09	26.99	26.12
453.povray	21.84	15.00	13.85
454.calculix	10.04	9.85	9.36
459.GemsFDTD	23.02	22.14	21.51
465.tonto	17.15	14.77	13.33
470.lbm	33.43	33.19	33.34
481.wrf	21.36	19.53	18.22
482.sphinx3	27.22	26.56	24.54

Table 13: 64-bits x86 Benchmark Ratios for DynamoRIO and Pin

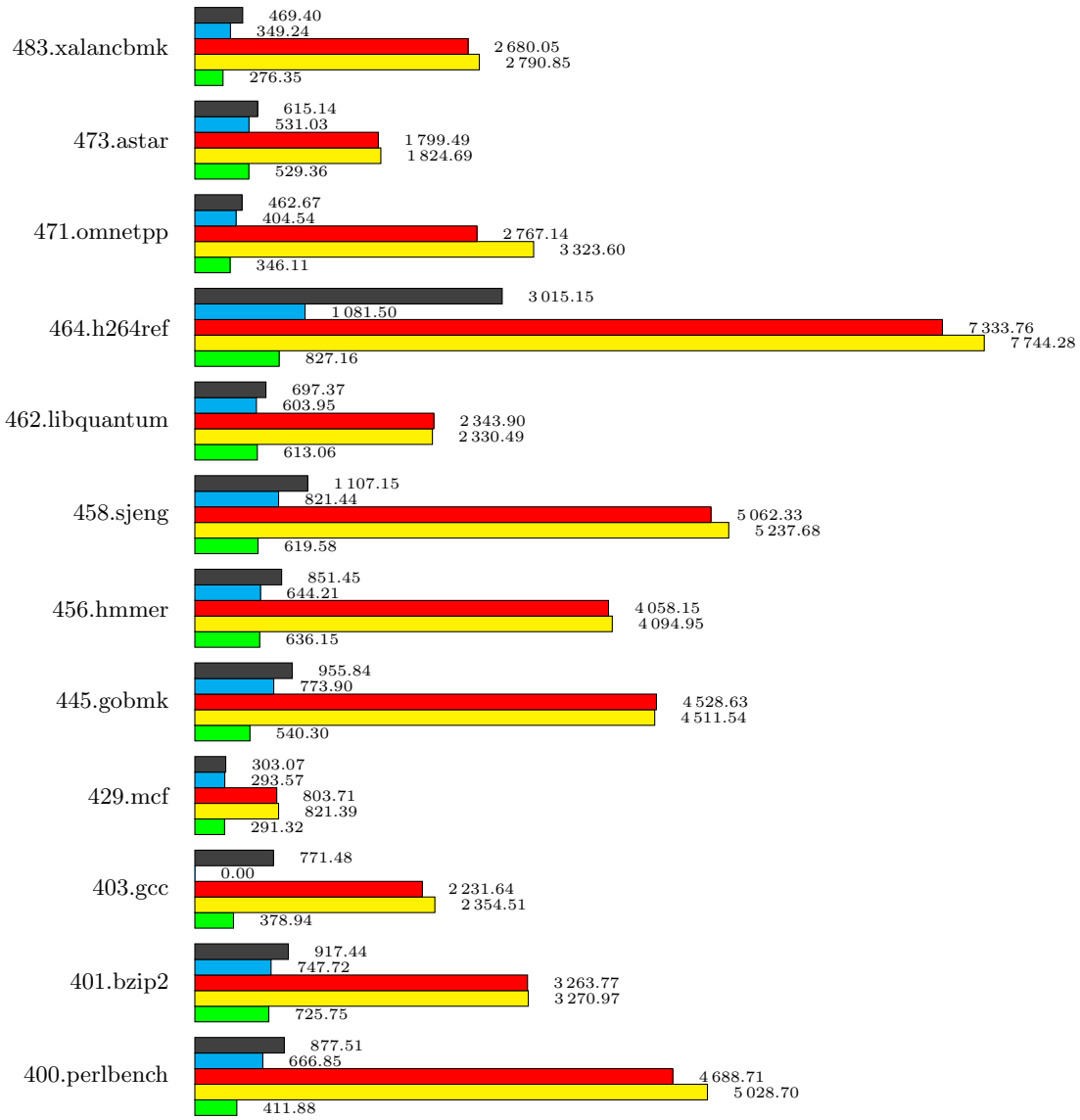

Figure 1: Run Times of 32-bits x86 specint

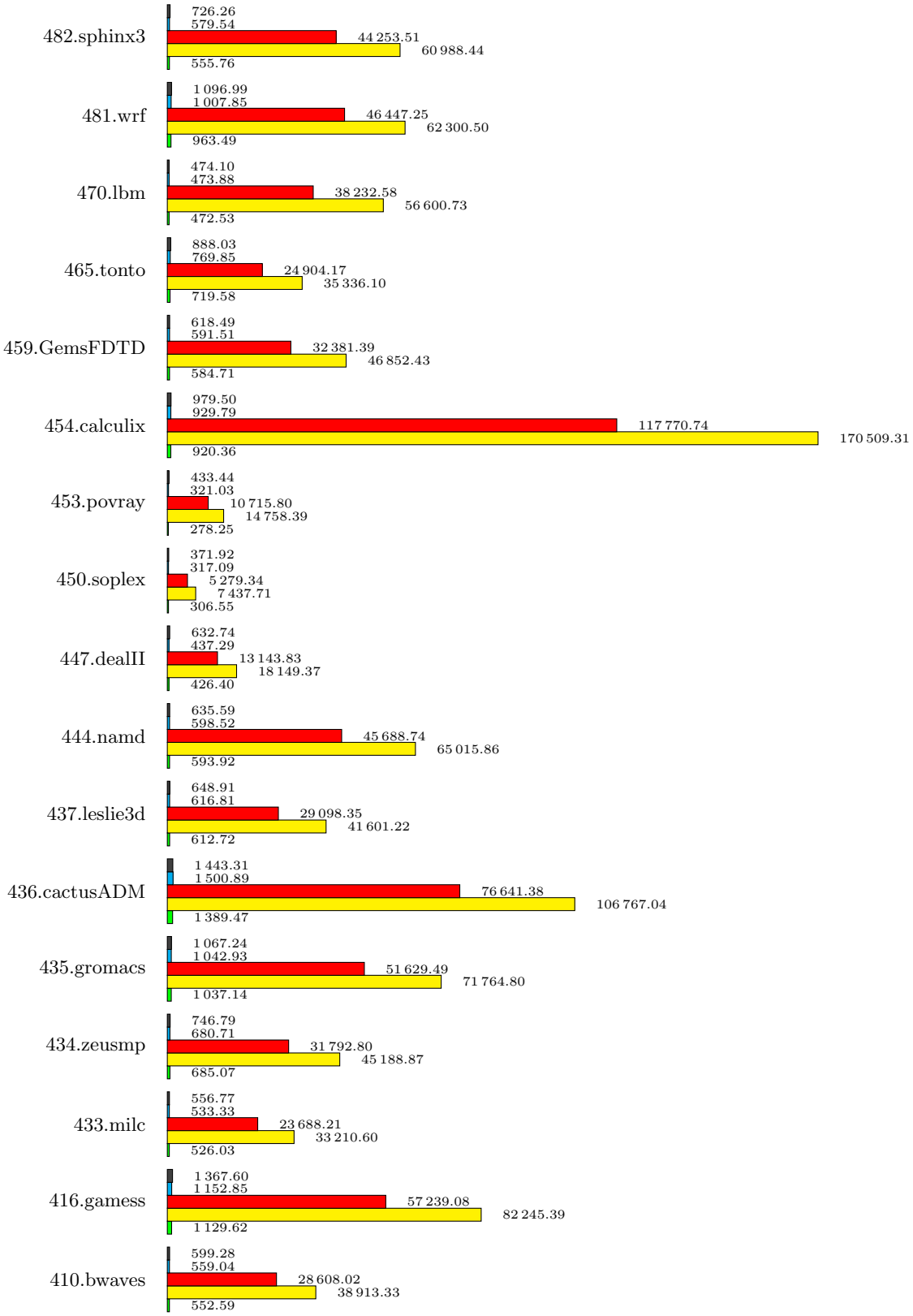

Figure 2: Run Times of 32-bits x86 specfp

32-bits Native
32-bits qemu-i386
64-bits qemu-i386
DynamoRIO
Pin

The user-mode QEMU is, therefore, much slower than the average [**?**]. Its slowdown is summarized on the next chart, whose bars represent the $\frac{NativeRunTime}{EmulatedRunTime}$ ratio, that is, how much slower the given emulation is in comparison to native runs.

Afterwards, the causes of QEMU's slowness are depicted. The following tables and bar charts are result of profiling runs done with the `perf` tool. The tables specify how much time was spent on each function that was running due to the given command, whereas the bar charts display the time in a more elaborated manner, grouping the application in its most meaningful areas: Benchmark, that is, the benchmark itself, on QEMU's code cache; Mutex, operations used by QEMU to manage critical sections; SoftFloat, floating-point operations done by QEMU using software; Qemu, any other QEMU emulation code; Perf, the profiling tool. Details of criteria used to make these areas were given in section 2.2. Figure **??** sums up QEMU profiling in one chart. Each column represents a benchmark runtime, divided according to time spent on each of the areas mentioned early. Next, each benchmark runtime is detailed with a table showing the most time-consuming functions and with a chart depicting the time spent on each of the areas referred early. "[.]" means user space, while "[k]" is kernel space. Analyzing these results, we can see that the emulations with the worst overhead performed a lot of mutex or floating point operations. More specifically, INT benchmarks had a considerable overhead due to mutex operations, while all FP benchmarks had a great chunk of their time spent on floating-point emulation. In fact, the mutex operations demand a lot of time, as they must sync threads to ensure correct execution of critical sections. Also, as stated in source code, QEMU uses SoftFloat IEC/IEEE Floating-point Arithmetic Package, implying that floating point operations are emulated by software, which justifies QEMU's enormous floating-point overhead, as one single hardware floating-point operation becomes a whole software routine.

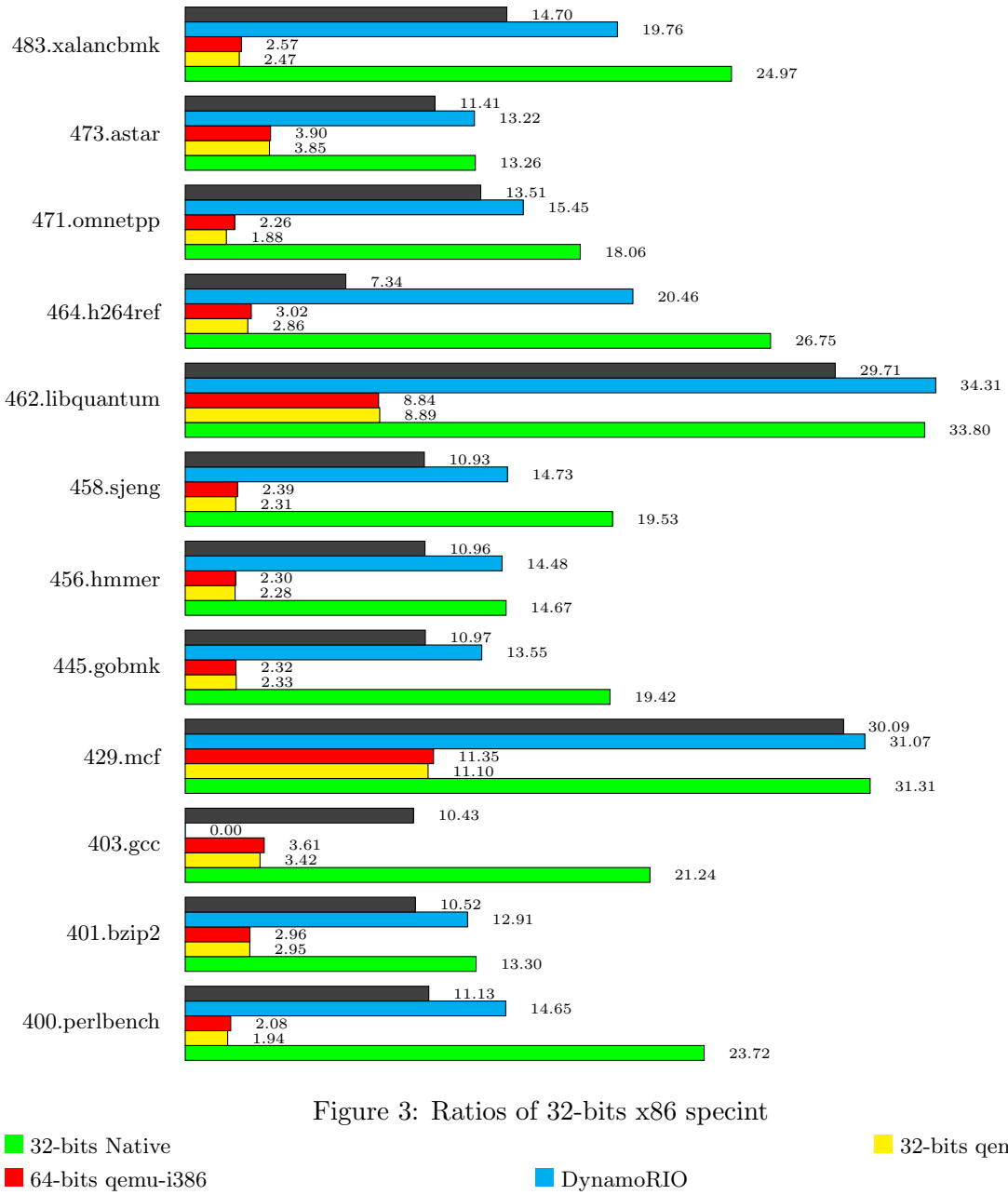

Figure 3: Ratios of 32-bits x86 specint

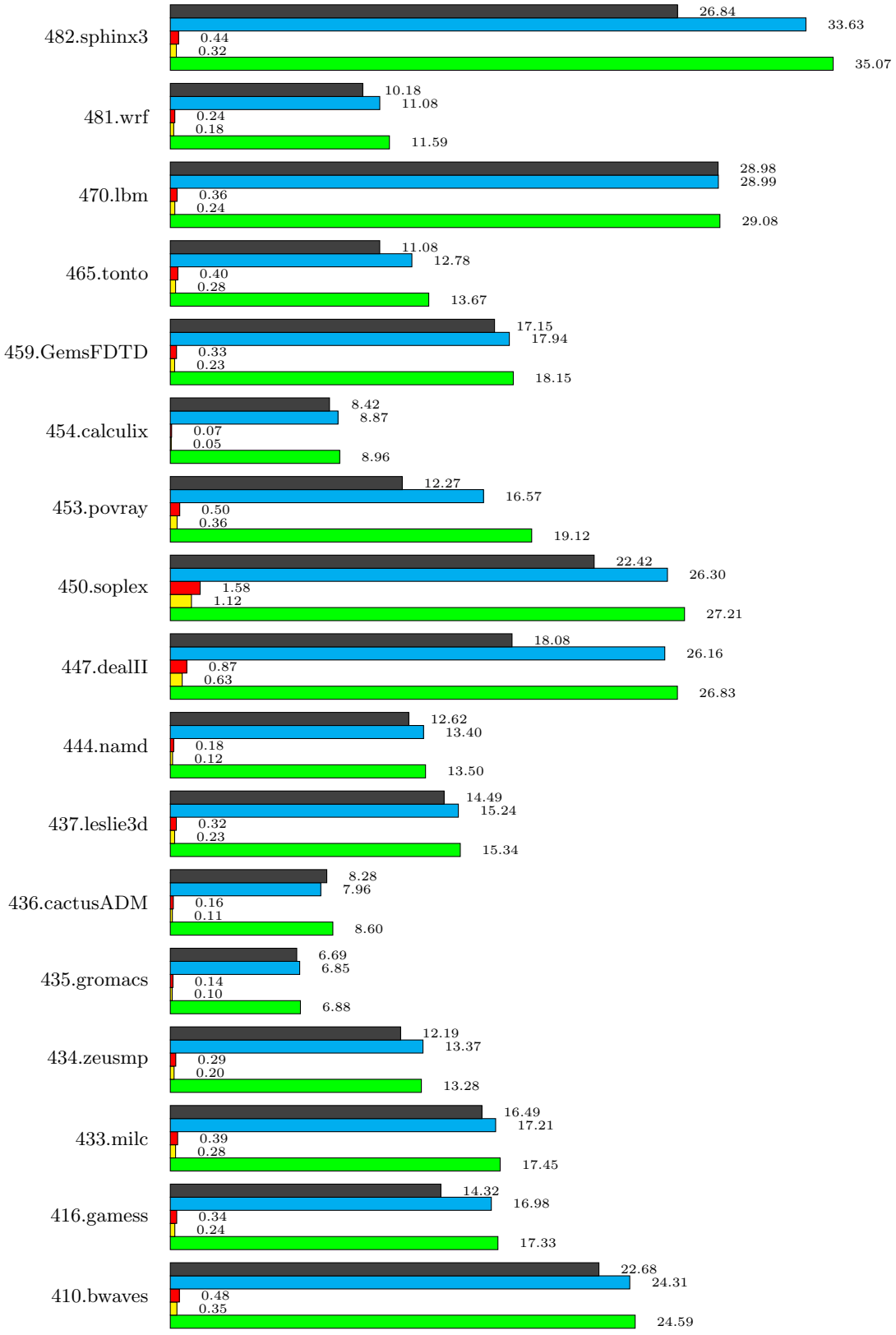

Figure 4: Ratios of 32-bits x86 specfp

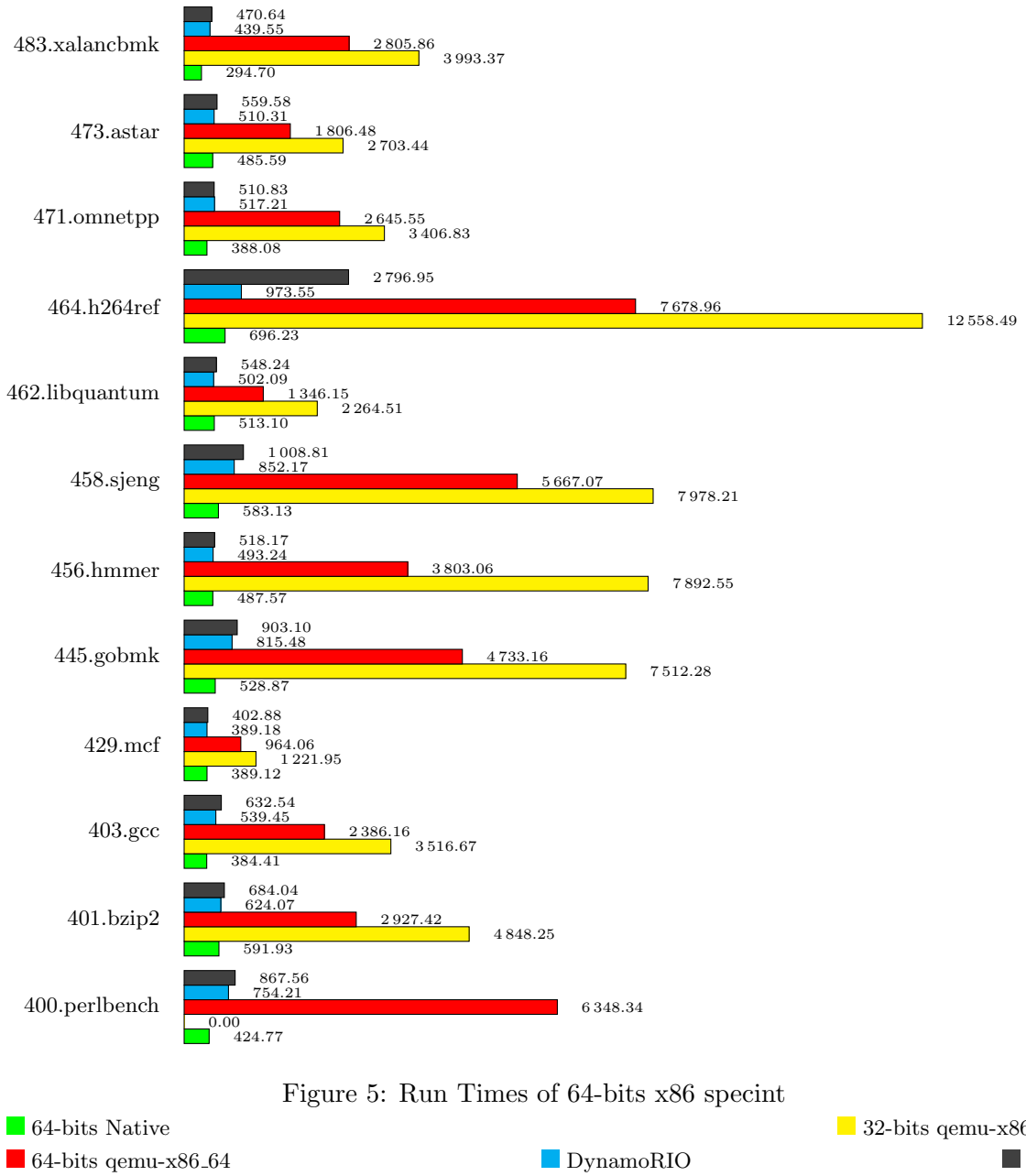

Figure 5: Run Times of 64-bits x86 specint

64-bits Native
32-bits qemu-x86_64
64-bits qemu-x86_64
DynamoRIO
Pin

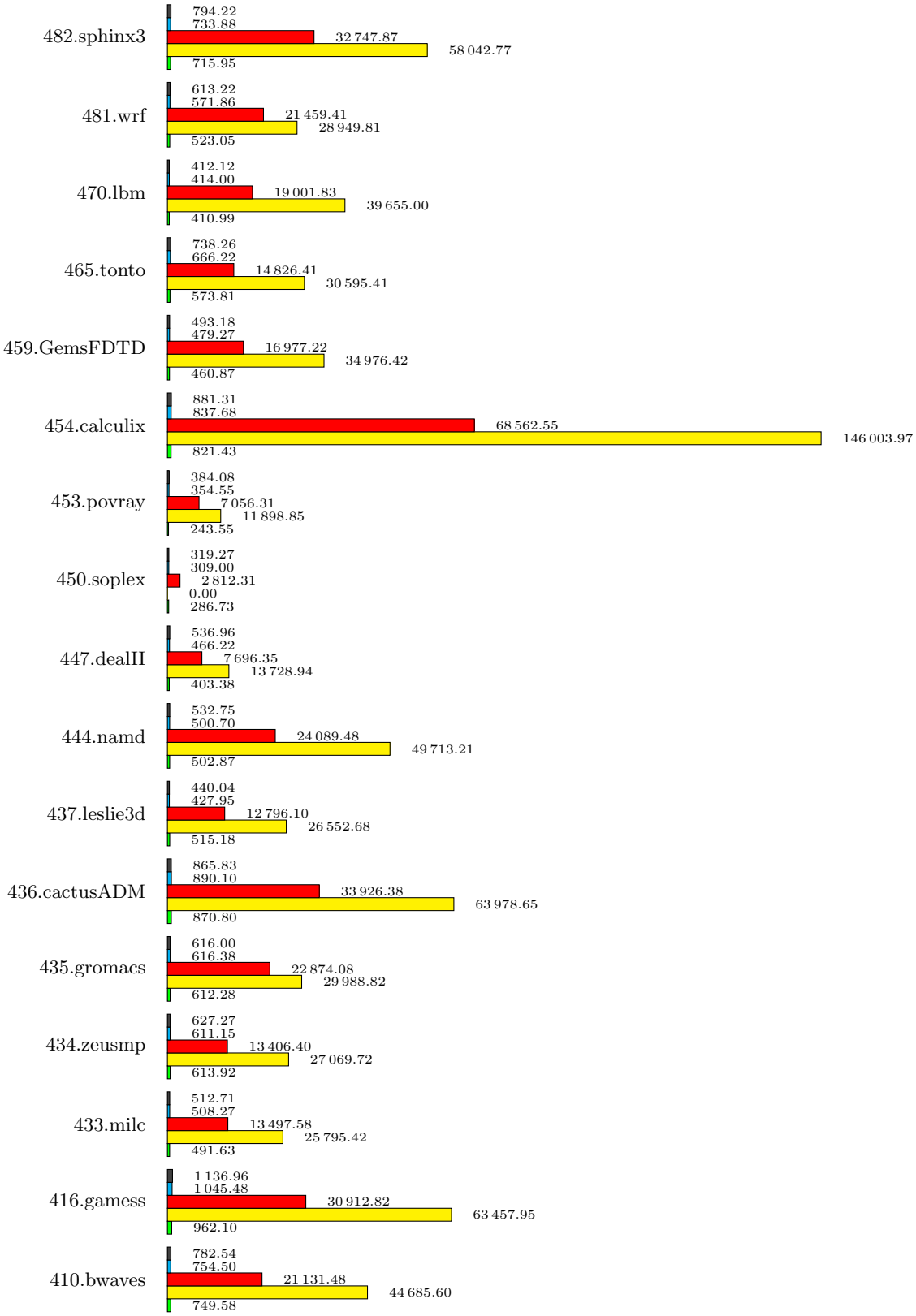

Figure 6: Run Times of 64-bits x86 specfp

64-bits Native
32-bits qemu-x86_64
64-bits qemu-x86_64
DynamoRIO
Pin

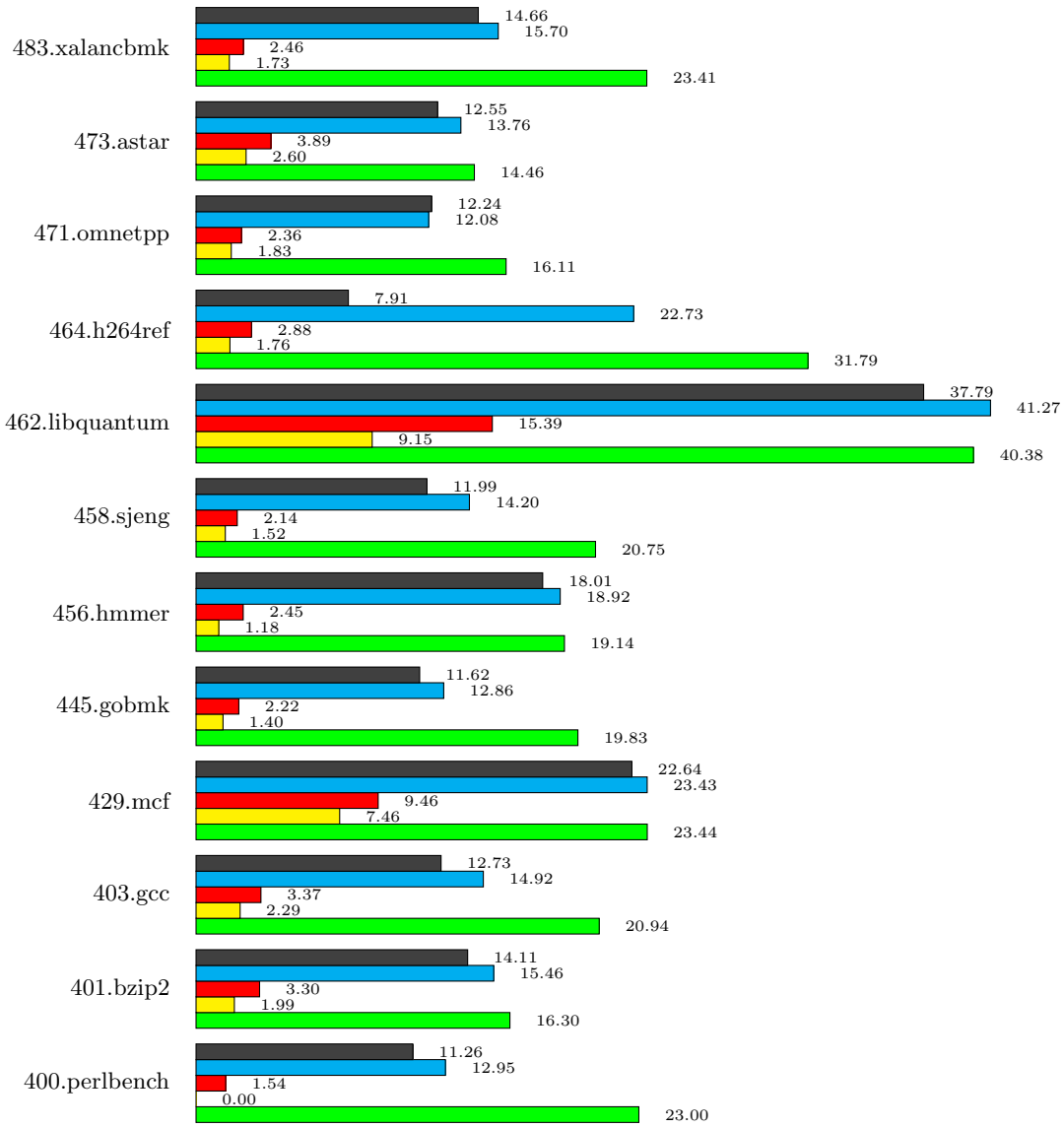

Figure 7: Ratios of 64-bits x86 specint

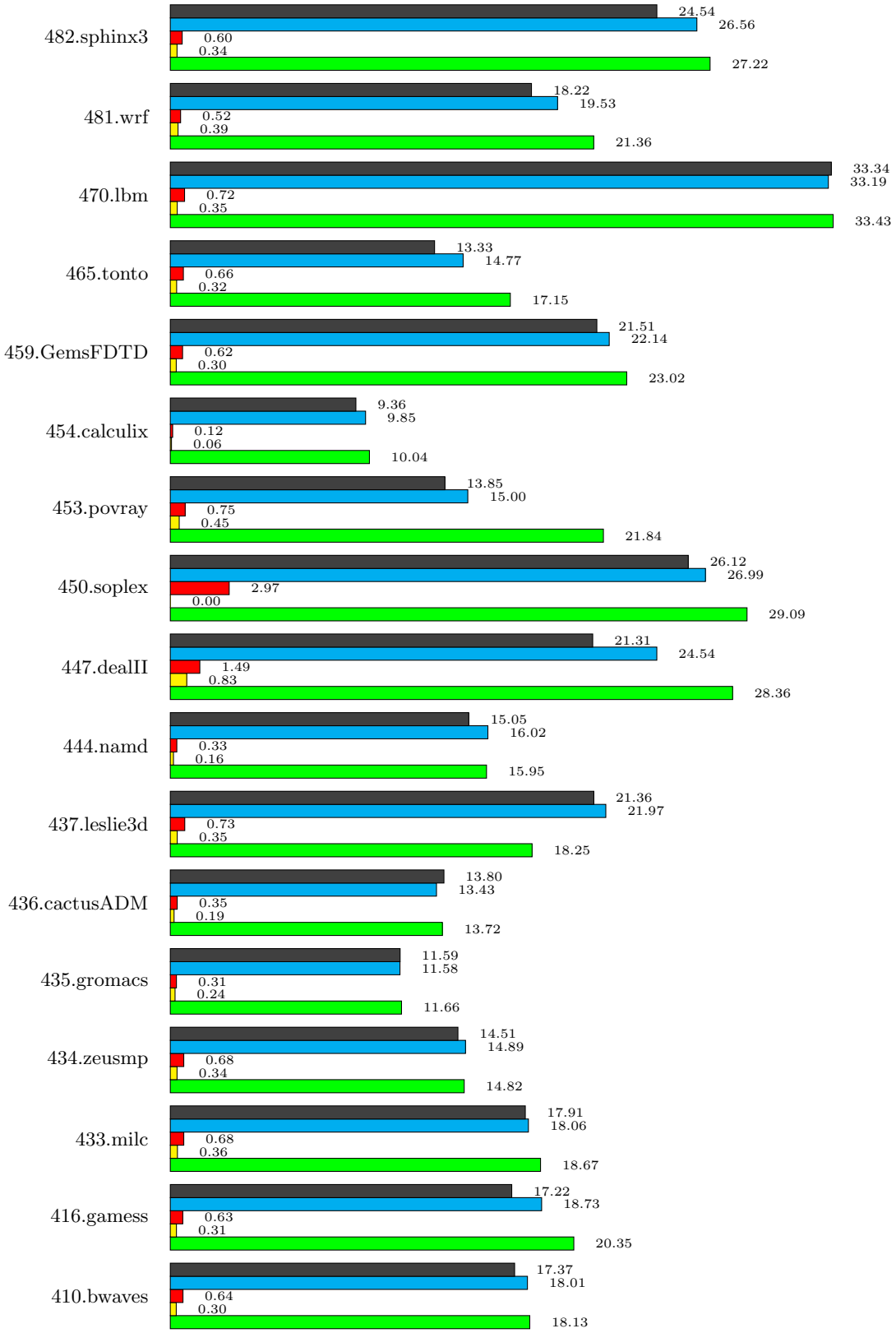

Figure 8: Ratios of 64-bits x86 specfp

64-bits Native
32-bits qemu-x86_64
64-bits qemu-x86_64
DynamoRIO
Pin

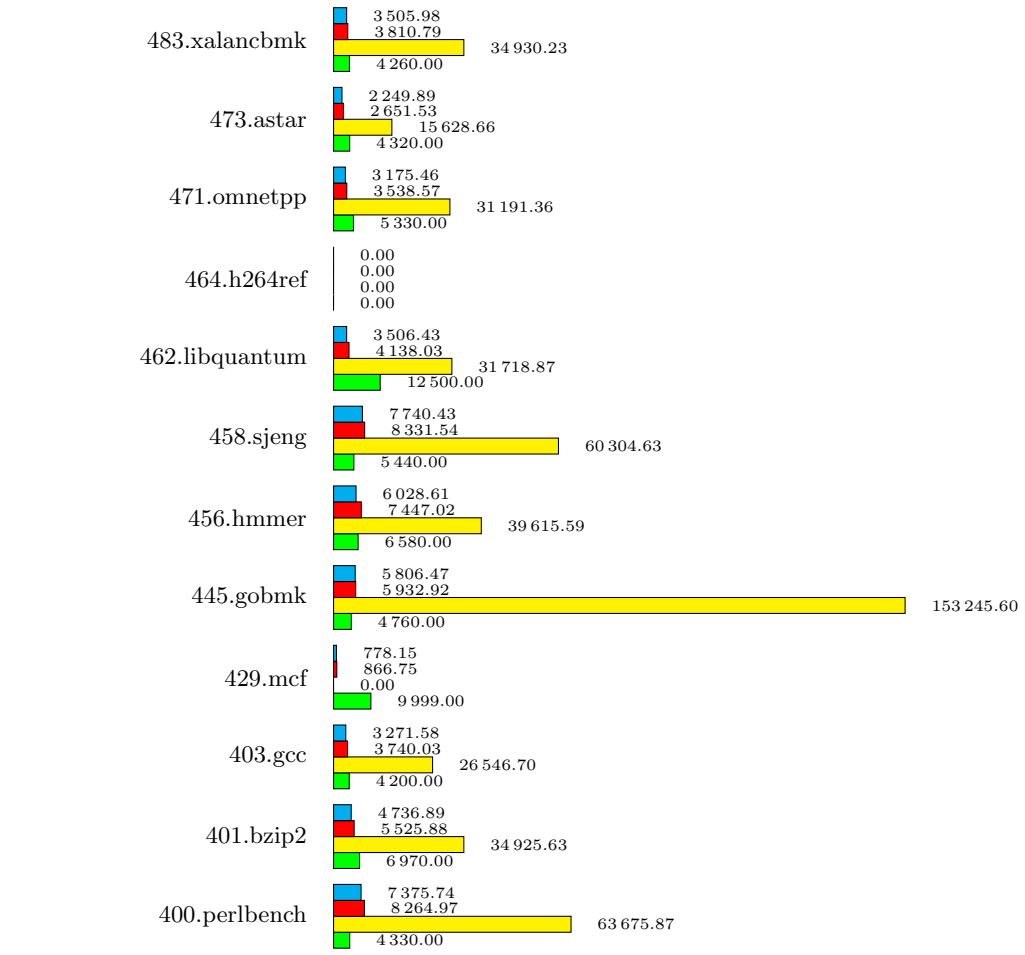

Figure 9: Run Times of Specint

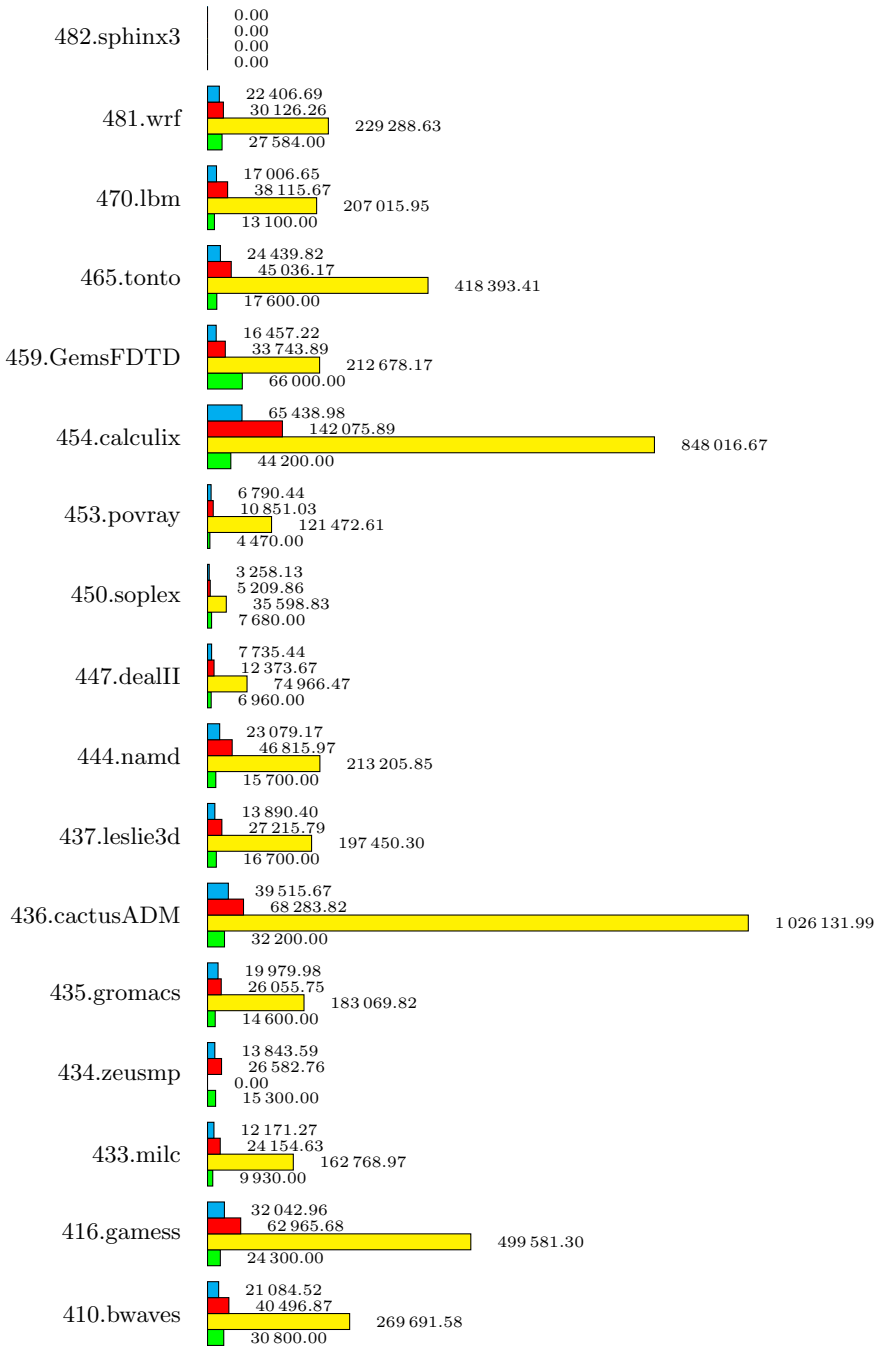

Figure 10: Run Times of Specfp

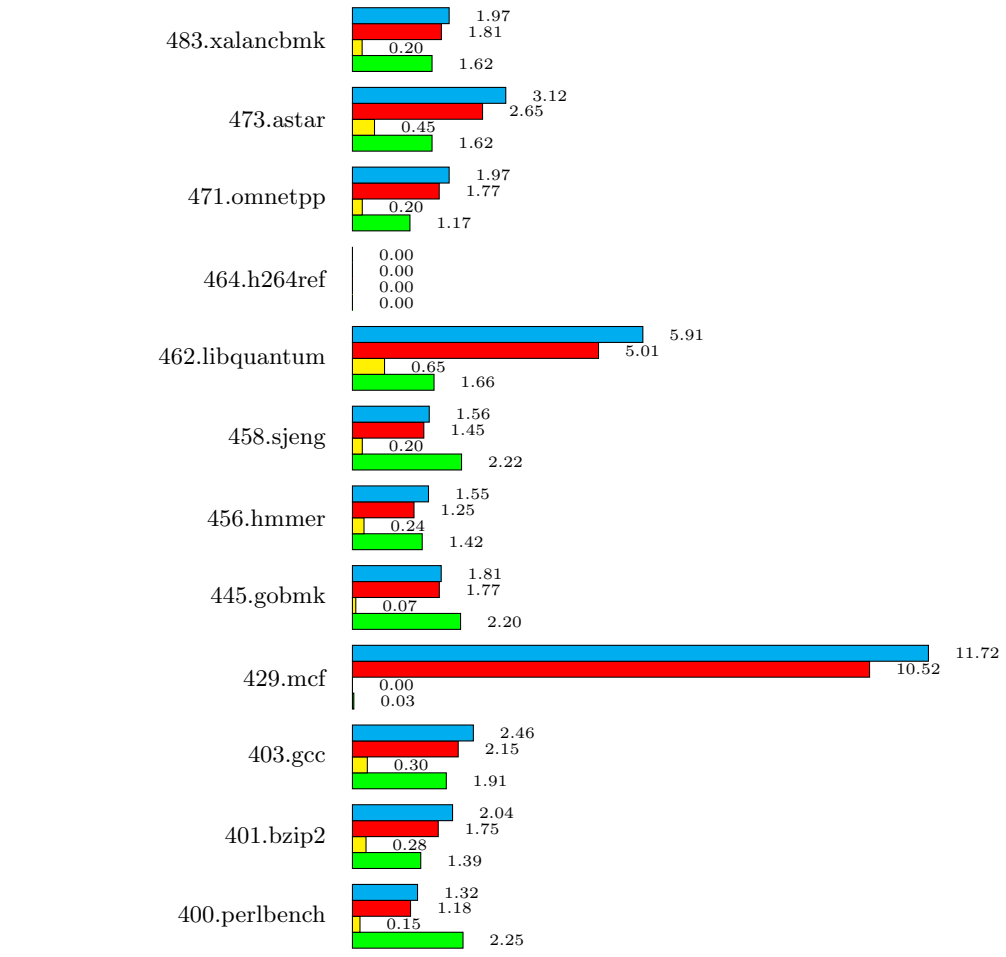

Figure 11: Ratios of Specint

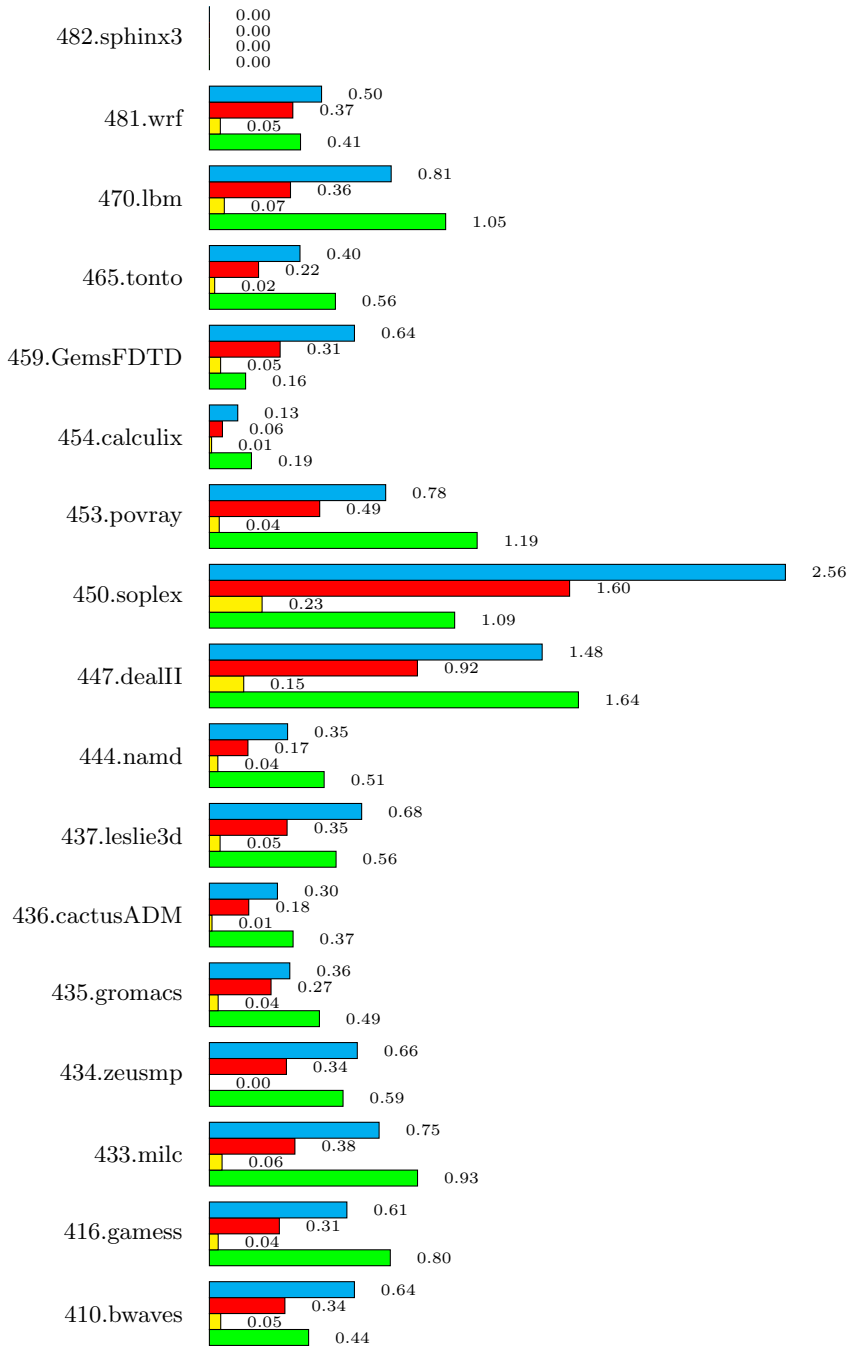

Figure 12: Ratios of Specfp

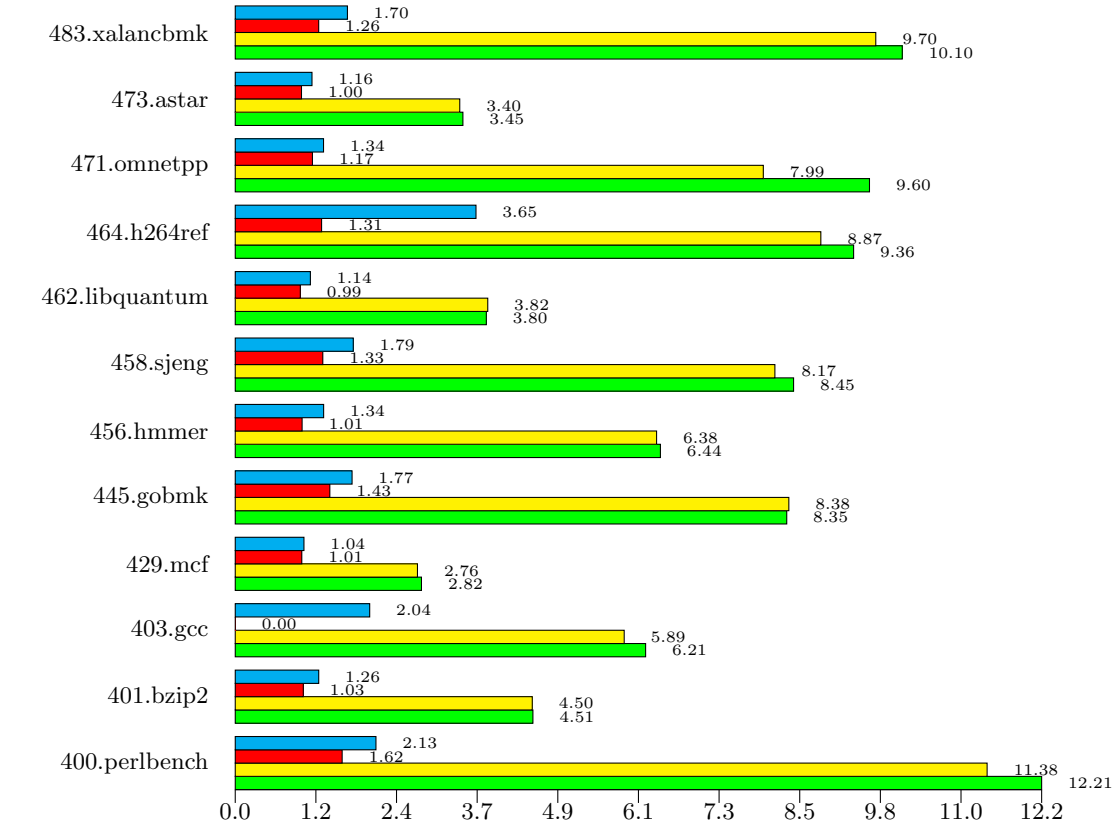

Figure 13: Overhead Specint for 32-bits x86

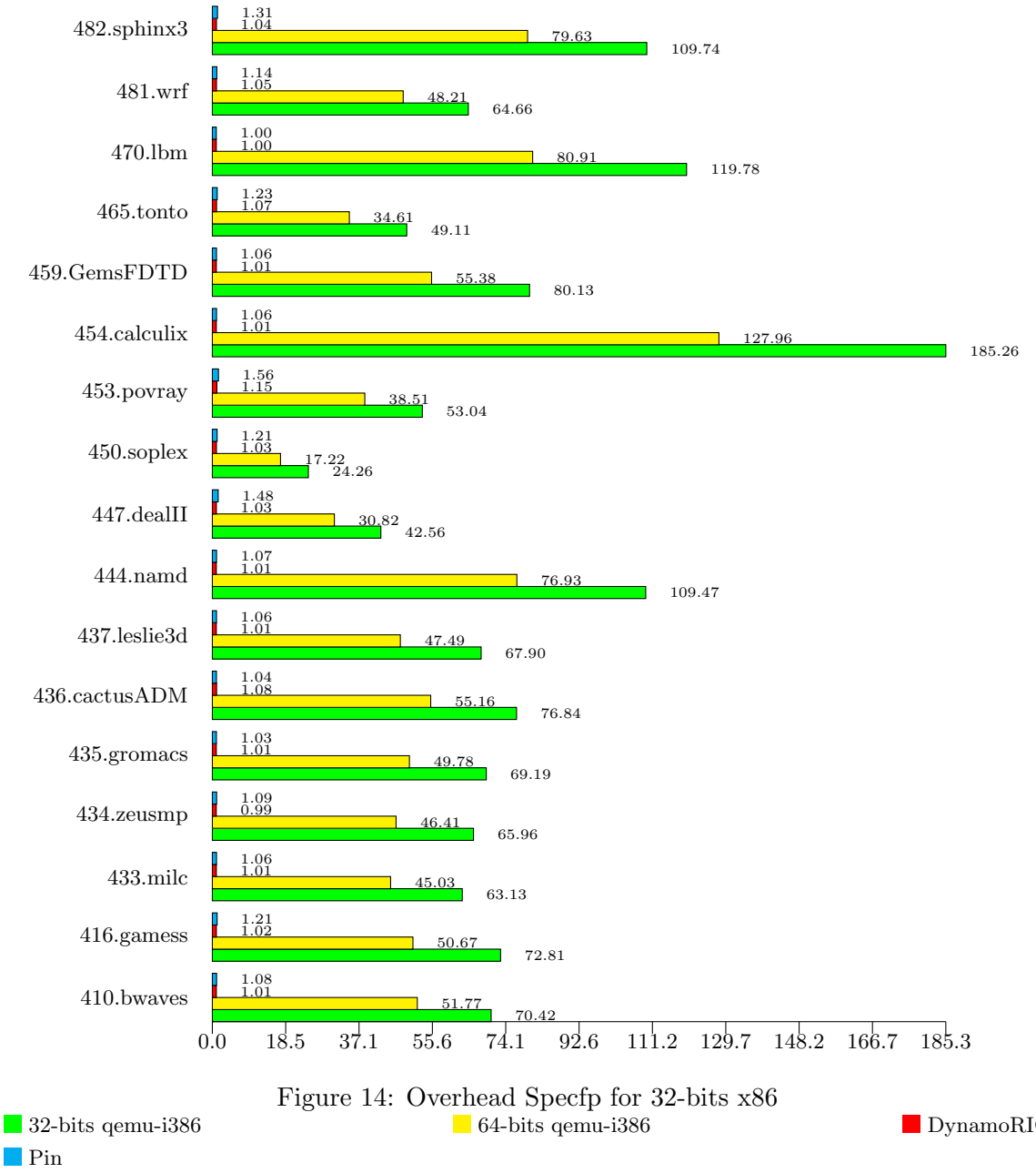

Figure 14: Overhead Specfp for 32-bits x86

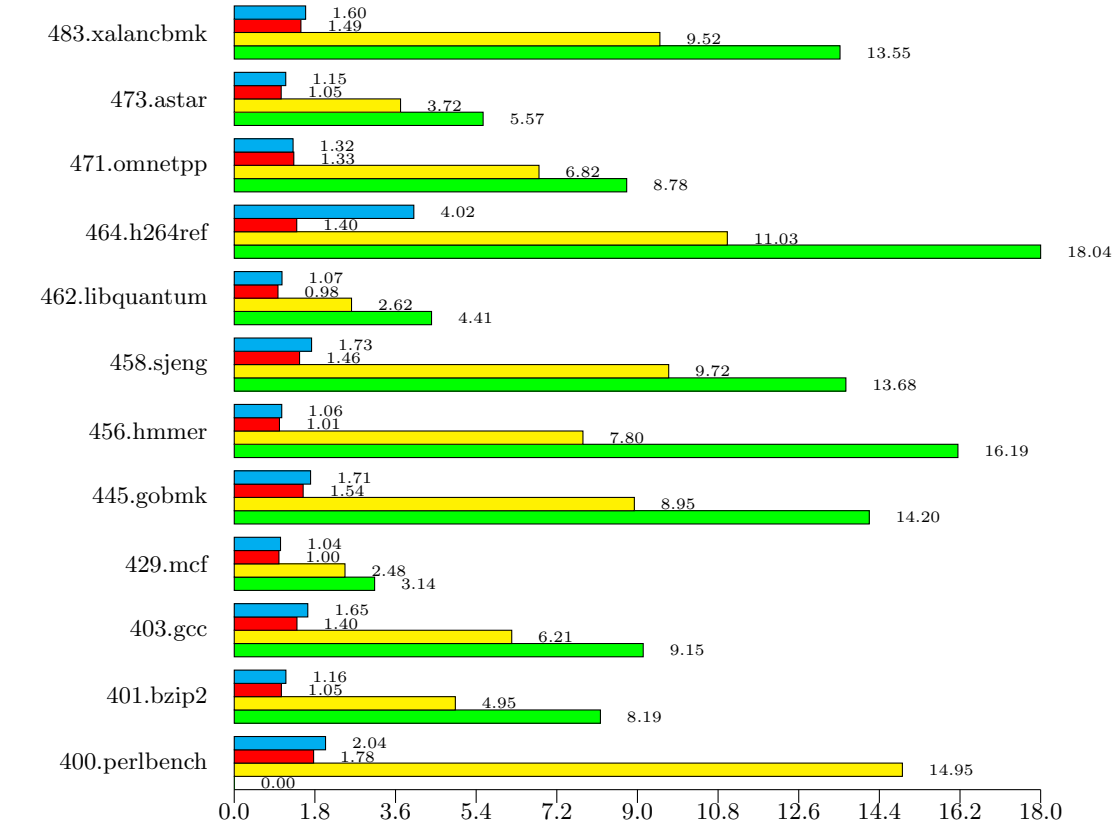

Figure 15: Overhead Specint for 64-bits x86

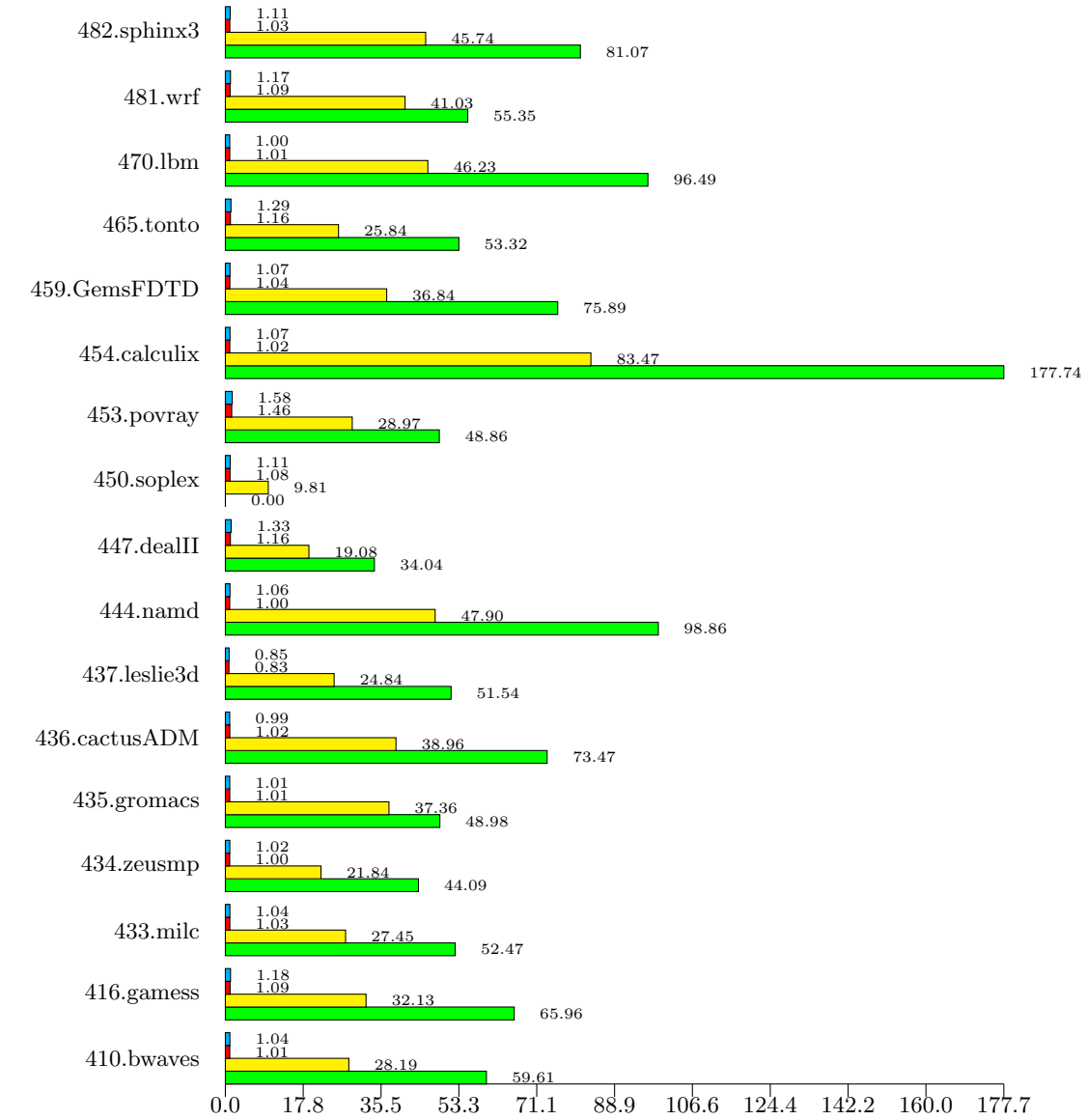

Figure 16: Overhead Specfp for 64-bits x86

Overhead(%)	Command	Symbol
51.61	qemu-x86_64	[.] 0x0000000083b9c3
18.41	qemu-x86_64	[.] cpu_x86_exec
14.10	qemu-x86_64	[.] __pthread_mutex_unlock
11.78	qemu-x86_64	[.] __pthread_mutex_lock
0.53	qemu-x86_64	[.] helper_cc_compute_all
0.20	qemu-x86_64	[.] helper_cc_compute_c
0.14	qemu-x86_64	[.] helper_pcmpeqb_xmm
0.13	qemu-x86_64	[.] float64_compare_quiet
0.13	qemu-x86_64	[.] tcg_reg_alloc_op
0.12	qemu-x86_64	[.] temp_save
0.12	qemu-x86_64	[.] float64_to_int64_round_to_zero
0.12	qemu-x86_64	[.] tcg_liveness_analysis
0.11	qemu-x86_64	[.] tcg_optimize
0.10	qemu-x86_64	[k] clear_page_c
0.10	qemu-x86_64	[.] estimateDiv128To64
0.09	qemu-x86_64	[.] helper_ucomisd
0.09	qemu-x86_64	[.] roundAndPackFloat64
0.07	qemu-x86_64	[.] disas_insn
0.06	qemu-x86_64	[.] float64_mul
0.05	qemu-x86_64	[.] helper_pmovmskb_xmm

Table 14: Profiling of 400.perlbench

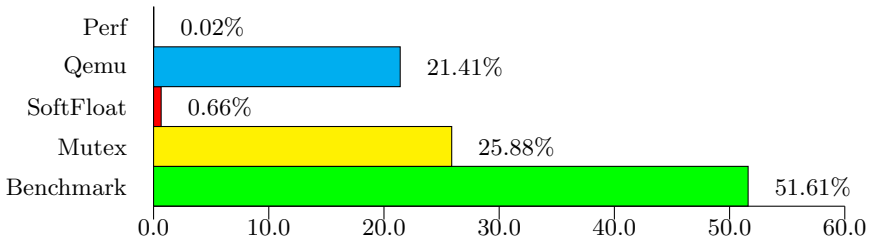

Figure 17: Main Code Areas for 400.perlbench

Overhead(%)	Command	Symbol
49.62	qemu-x86_64	[.] 0x0000000083ee8a
34.43	qemu-x86_64	[.] 0x0000000086aa70
4.64	qemu-x86_64	[.] __pthread_mutex_unlock
4.27	qemu-x86_64	[.] cpu_x86_exec
3.95	qemu-x86_64	[.] __pthread_mutex_lock
2.52	qemu-x86_64	[.] helper_cc_compute_all
0.38	qemu-x86_64	[.] 0x000000008ab5ff
0.01	qemu-x86_64	[k] clear_page_c
0.01	qemu-x86_64	[k] copy_user_generic_string

Table 15: Profiling of 401.bzip2

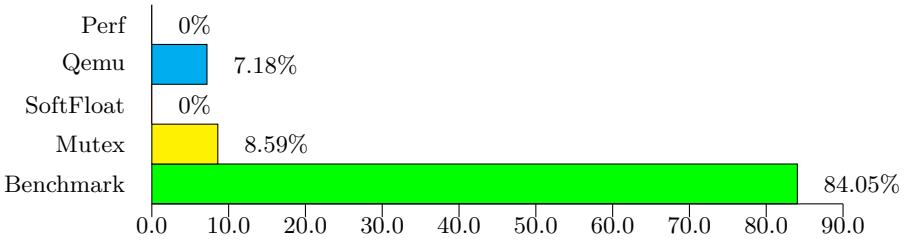

Figure 18: Main Code Areas for 401.bzip2

Overhead(%)	Command	Symbol
41.01	qemu-x86_64	[.] 0x00000000cebe6a
23.26	qemu-x86_64	[.] 0x0000000084a937
14.68	qemu-x86_64	[.] 0x000000004f378d
7.41	qemu-x86_64	[.] cpu_x86_exec
5.78	qemu-x86_64	[.] __pthread_mutex_unlock
4.88	qemu-x86_64	[.] __pthread_mutex_lock
1.52	qemu-x86_64	[.] 0x00000000d2517d
0.32	qemu-x86_64	[.] helper_cc_compute_all
0.24	qemu-x86_64	[.] div64
0.14	qemu-x86_64	[.] helper_cc_compute_c
0.11	qemu-x86_64	[.] helper_idivl_EAX
0.06	qemu-x86_64	[.] helper_divq_EAX
0.04	qemu-x86_64	[.] helper_psrldq_xmm
0.04	qemu-x86_64	[k] clear_page_c
0.04	qemu-x86_64	[.] helper_pshufd_xmm
0.02	qemu-x86_64	[.] helper_imulq_T0_T1
0.01	qemu-x86_64	[k] page_fault
0.01	qemu-x86_64	[.] helper_punpckldq_xmm
0.01	qemu-x86_64	[.] helper_pcmpeqb_xmm
0.01	qemu-x86_64	[.] tcg_reg_alloc_op

Table 16: Profiling of 403.gcc

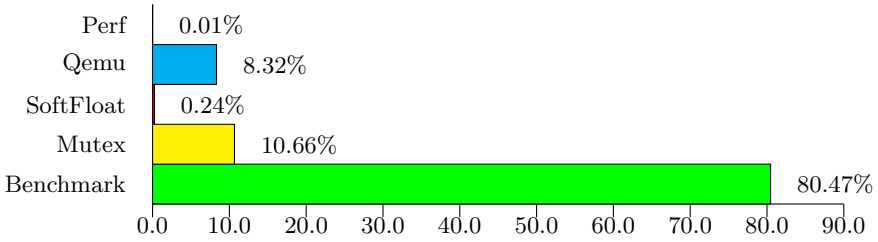

Figure 19: Main Code Areas for 403.gcc

Overhead(%)	Command	Symbol
91.55	qemu-x86_64	[.] 0x00000000851c25
6.32	qemu-x86_64	[.] helper_cc_compute_all
0.72	qemu-x86_64	[.] __pthread_mutex_unlock
0.59	qemu-x86_64	[.] __pthread_mutex_lock
0.56	qemu-x86_64	[.] cpu_x86_exec
0.02	qemu-x86_64	[k] clear_page_c
0.01	perf_2.6.32-46	[k] copy_user_generic_string
0.01	qemu-x86_64	[k] page_fault
0.01	qemu-x86_64	[k] __ticket_spin_lock
0.01	qemu-x86_64	[k] hrtimer_interrupt

Table 17: Profiling of 429.mcf

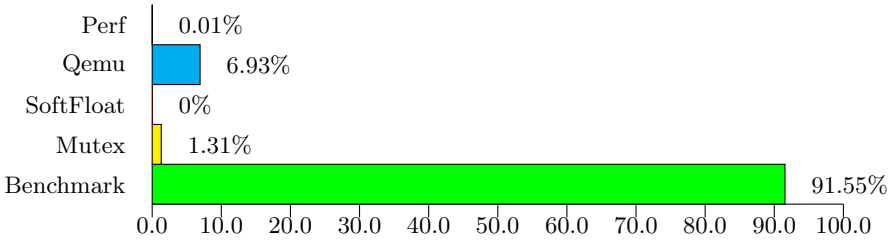

Figure 20: Main Code Areas for 429.mcf

Overhead(%)	Command	Symbol
45.57	:1418	[.] 0x000000008a8c20
32.79	qemu-x86_64	[.] 0x000000028850bf
8.36	qemu-x86_64	[.] cpu_x86_exec
5.59	qemu-x86_64	[.] __pthread_mutex_unlock
4.75	qemu-x86_64	[.] __pthread_mutex_lock
1.62	qemu-x86_64	[.] helper_imulq_T0_T1
0.56	qemu-x86_64	[.] helper_cc_compute_all
0.09	qemu-x86_64	[.] helper_cc_compute_c
0.06	qemu-x86_64	[.] float32_compare_quiet
0.05	qemu-x86_64	[.] helper_ucomiss
0.05	qemu-x86_64	[.] div64
0.04	qemu-x86_64	[.] int32_to_float64
0.04	qemu-x86_64	[.] float64_compare_quiet
0.03	qemu-x86_64	[.] roundAndPackFloat32
0.03	qemu-x86_64	[.] helper_idivl_EAX
0.03	qemu-x86_64	[.] helper_ucomisd
0.02	qemu-x86_64	[.] float32_mul
0.02	qemu-x86_64	[.] helper_divq_EAX
0.01	qemu-x86_64	[.] float32_div
0.01	qemu-x86_64	[.] helper_punpckldq_xmm

Table 18: Profiling of 445.gobmk

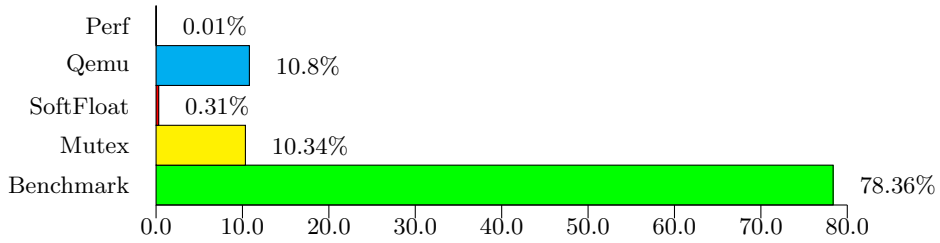

Figure 21: Main Code Areas for 445.gobmk

Overhead(%)	Command	Symbol
87.06	qemu-x86_64	[.] 0x007f6a2b484cd9
1.10	qemu-x86_64	[.] roundAndPackFloat32
1.06	qemu-x86_64	[.] __pthread_mutex_unlock
1.02	qemu-x86_64	[.] cpu_x86_exec
0.98	qemu-x86_64	[.] helper_pcmpgtl_xmm
0.88	qemu-x86_64	[.] __pthread_mutex_lock
0.86	qemu-x86_64	[.] estimateDiv128To64
0.77	qemu-x86_64	[.] addFloat32Sigs
0.67	qemu-x86_64	[.] helper_paddl_xmm
0.53	qemu-x86_64	[.] float32_compare_quiet
0.52	qemu-x86_64	[.] helper_pcmpeqb_xmm
0.39	qemu-x86_64	[.] helper_ucomiss
0.34	qemu-x86_64	[.] helper_por_xmm
0.30	qemu-x86_64	[.] helper_pmovmskb_xmm
0.29	qemu-x86_64	[.] helper_pandn_xmm
0.29	qemu-x86_64	[.] helper_cc_compute_all
0.29	qemu-x86_64	[.] helper_pand_xmm
0.29	qemu-x86_64	[.] helper_psrldq_xmm
0.27	qemu-x86_64	[.] helper_punpcklbw_xmm
0.25	qemu-x86_64	[.] roundAndPackFloat64

Table 19: Profiling of 456.hmmer

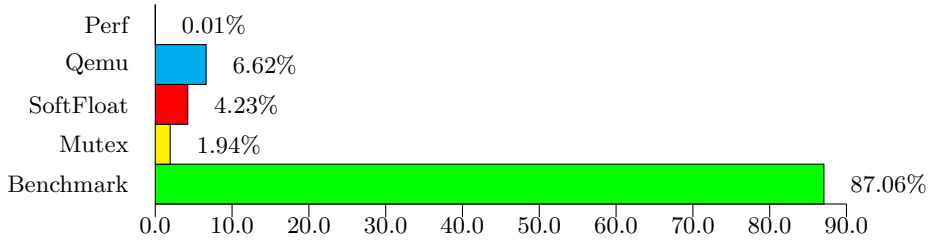

Figure 22: Main Code Areas for 456.hmmer

Overhead(%)	Command	Symbol
59.15	qemu-x86_64	[.] 0x0000000084c62d
16.46	qemu-x86_64	[.] cpu_x86_exec
12.76	qemu-x86_64	[.] __pthread_mutex_unlock
10.66	qemu-x86_64	[.] __pthread_mutex_lock
0.38	qemu-x86_64	[.] helper_divl_EAX
0.36	qemu-x86_64	[.] helper_cc_compute_all
0.05	qemu-x86_64	[.] helper_cc_compute_c
0.01	perf_2.6.32-46	[k] copy_user_generic_string

Table 20: Profiling of 458.sjeng

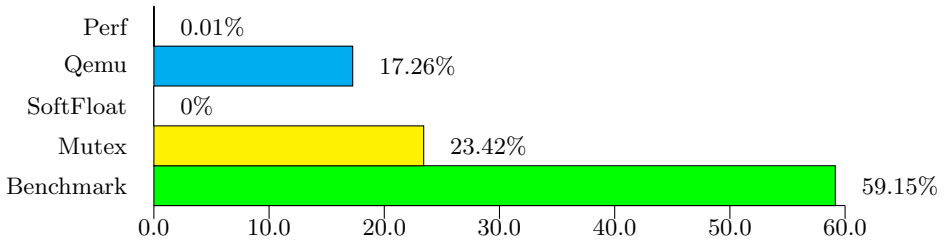

Figure 23: Main Code Areas for 458.sjeng

Overhead(%)	Command	Symbol
93.65	qemu-x86_64	[.] 0x00000000851913
1.39	qemu-x86_64	[.] __pthread_mutex_unlock
1.18	qemu-x86_64	[.] __pthread_mutex_lock
1.16	qemu-x86_64	[.] cpu_x86_exec
0.48	qemu-x86_64	[.] roundAndPackFloat32
0.41	qemu-x86_64	[.] float32_mul
0.15	qemu-x86_64	[.] helper_cc_compute_all
0.15	qemu-x86_64	[.] estimateDiv128To64
0.14	qemu-x86_64	[.] addFloat32Sigs
0.11	qemu-x86_64	[.] float32_div
0.11	qemu-x86_64	[.] helper_mulss
0.10	qemu-x86_64	[.] estimateSqrt32
0.08	qemu-x86_64	[.] helper_punpckldq_xmm
0.07	qemu-x86_64	[.] roundAndPackFloat64
0.06	qemu-x86_64	[.] float64_sqrt
0.06	qemu-x86_64	[.] addFloat64Sigs
0.05	qemu-x86_64	[.] float32_to_float64
0.05	qemu-x86_64	[.] subFloat32Sigs
0.03	qemu-x86_64	[.] float32_compare_quiet
0.03	qemu-x86_64	[.] float32_add

Table 21: Profiling of 462.libquantum

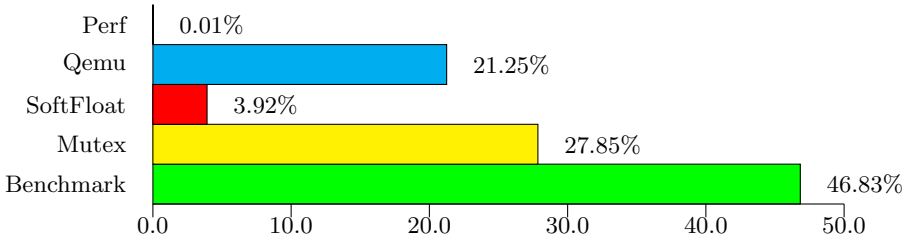

Figure 24: Main Code Areas for 462.libquantum

Overhead(%)	Command	Symbol
66.42	qemu-x86_64	[.] 0x007fad743cfca4
11.54	qemu-x86_64	[.] __pthread_mutex_unlock
10.74	qemu-x86_64	[.] cpu_x86_exec
9.62	qemu-x86_64	[.] __pthread_mutex_lock
0.47	qemu-x86_64	[.] helper_idivl_EAX
0.19	qemu-x86_64	[.] helper_cc_compute_all
0.15	qemu-x86_64	[.] helper_divl_EAX
0.09	qemu-x86_64	[.] helper_paddl_xmm
0.09	qemu-x86_64	[.] helper_imulq_T0_T1
0.07	qemu-x86_64	[.] helper_punpcklwd_xmm
0.07	qemu-x86_64	[.] roundAndPackFloat64
0.05	qemu-x86_64	[.] float64_mul
0.05	qemu-x86_64	[.] helper_punpckhwd_xmm
0.05	qemu-x86_64	[.] addFloat64Sigs
0.03	qemu-x86_64	[.] float64_compare_quiet
0.03	qemu-x86_64	[.] subFloat64Sigs
0.02	qemu-x86_64	[.] helper_ucomisd
0.02	qemu-x86_64	[.] helper_cc_compute_c
0.02	qemu-x86_64	[.] float64_add
0.02	qemu-x86_64	[.] int32_to_float64

Table 22: Profiling of 464.h264ref

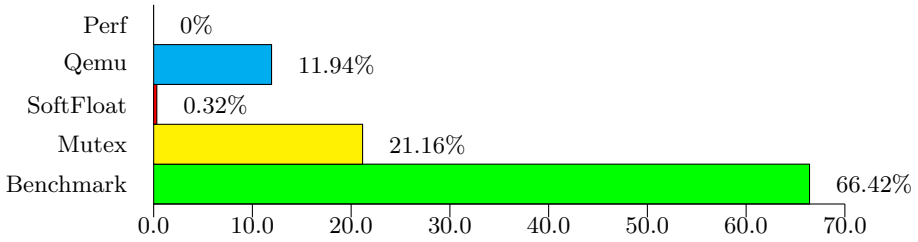

Figure 25: Main Code Areas for 464.h264ref

Overhead(%)	Command	Symbol
46.83	qemu-x86_64	[.] 0x007f0707cd9cd9
15.12	qemu-x86_64	[.] cpu_x86_exec
14.14	qemu-x86_64	[.] __pthread_mutex_unlock
13.71	qemu-x86_64	[.] __pthread_mutex_lock
2.60	qemu-x86_64	[.] float64_compare_quiet
1.85	qemu-x86_64	[.] helper_ucomisd
1.57	qemu-x86_64	[.] helper_cc_compute_all
1.34	qemu-x86_64	[.] helper_pcmpeqb_xmm
0.39	qemu-x86_64	[.] roundAndPackFloat64
0.36	qemu-x86_64	[.] helper_psubb_xmm
0.29	qemu-x86_64	[.] helper_bsf
0.28	qemu-x86_64	[.] helper_pmovmskb_xmm
0.27	qemu-x86_64	[.] addFloat64Sigs
0.16	qemu-x86_64	[.] div64
0.16	qemu-x86_64	[.] float64_mul
0.13	qemu-x86_64	[.] normalizeRoundAndPackFloat64
0.12	qemu-x86_64	[.] helper_cc_compute_c
0.11	qemu-x86_64	[.] helper_pslldq_xmm
0.08	qemu-x86_64	[.] float64_add
0.06	qemu-x86_64	[.] helper_idivq_EAX

Table 23: Profiling of 471.omnetpp

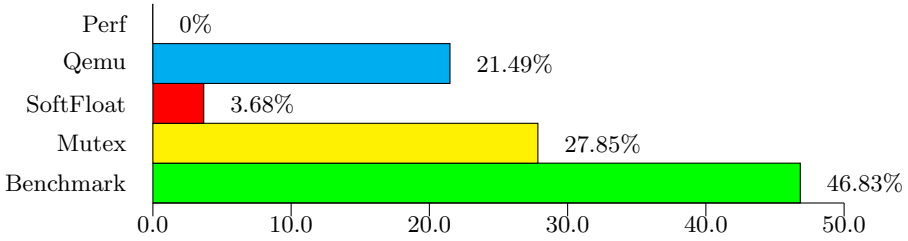

Figure 26: Main Code Areas for 471.omnetpp

Overhead(%)	**Command**	**Symbol**
81.91	qemu-x86_64	[.] 0x0000000083efd4
6.89	qemu-x86_64	[.] __pthread_mutex_unlock
5.78	qemu-x86_64	[.] __pthread_mutex_lock
4.96	qemu-x86_64	[.] cpu_x86_exec
0.23	qemu-x86_64	[.] helper_cc_compute_all
0.03	qemu-x86_64	[.] roundAndPackFloat32
0.02	qemu-x86_64	[.] float32_mul
0.01	qemu-x86_64	[k] copy_user_generic_string
0.01	qemu-x86_64	[.] addFloat32Sigs
0.01	qemu-x86_64	[.] helper_cc_compute_c
0.01	qemu-x86_64	[k] __ticket_spin_lock
0.01	qemu-x86_64	[.] normalizeRoundAndPackFloat32

Table 24: Profiling of 473.astar

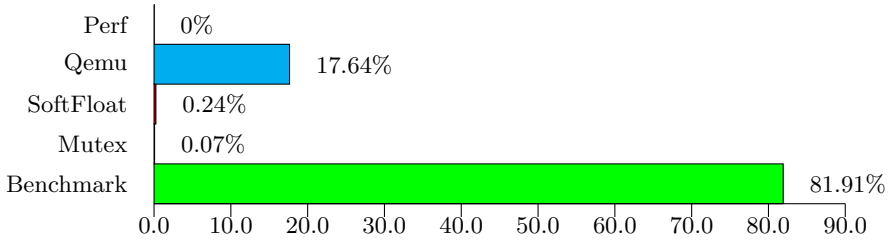

Figure 27: Main Code Areas for 473.astar

Overhead(%)	Command	Symbol
51.87	qemu-x86_64	[.] 0x007f4675f79745
19.66	qemu-x86_64	[.] cpu_x86_exec
14.48	qemu-x86_64	[.] __pthread_mutex_unlock
12.14	qemu-x86_64	[.] __pthread_mutex_lock
0.20	qemu-x86_64	[.] float64_compare_quiet
0.19	qemu-x86_64	[.] helper_imulq_T0_T1
0.17	qemu-x86_64	[.] helper_cc_compute_all
0.16	qemu-x86_64	[.] helper_ucomisd
0.16	qemu-x86_64	[.] roundAndPackFloat64
0.12	qemu-x86_64	[.] float64_to_int64_round_to_zero
0.10	qemu-x86_64	[.] addFloat64Sigs
0.09	qemu-x86_64	[.] helper_cc_compute_c
0.07	qemu-x86_64	[.] normalizeRoundAndPackFloat64
0.05	qemu-x86_64	[.] helper_cvtsq2sd
0.05	qemu-x86_64	[.] countLeadingZeros64
0.04	qemu-x86_64	[.] helper_pxor_xmm
0.04	qemu-x86_64	[.] subFloat64Sigs
0.04	qemu-x86_64	[.] int64_to_float64
0.03	qemu-x86_64	[.] helper_lock
0.03	qemu-x86_64	[.] helper_cvttsd2sq

Table 25: Profiling of 483.xalancbmk

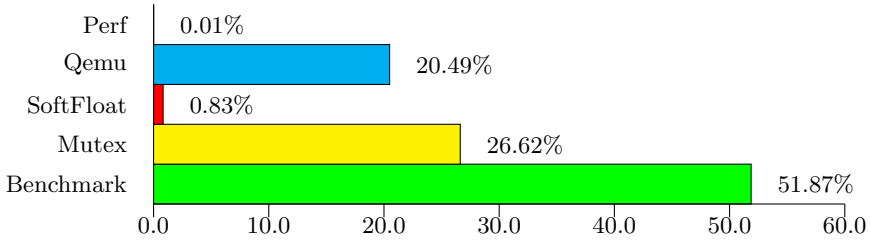

Figure 28: Main Code Areas for 483.xalancbmk

Overhead(%)	Command	Symbol
22.67	qemu-x86_64	[.] float64_mul
19.43	qemu-x86_64	[.] roundAndPackFloat64
17.47	qemu-x86_64	[.] 0x0000000083a850
8.29	qemu-x86_64	[.] addFloat64Sigs
5.67	qemu-x86_64	[.] subFloat64Sigs
3.94	qemu-x86_64	[.] float64_add
3.63	qemu-x86_64	[.] helper_mulsd
3.18	qemu-x86_64	[.] normalizeRoundAndPackFloat64
2.00	qemu-x86_64	[.] helper_addsd
1.57	qemu-x86_64	[.] estimateDiv128To64
0.64	qemu-x86_64	[.] float64_div
0.55	perf_2.6.32-46	[.] 0x0000000000c676
0.51	qemu-x86_64	[.] countLeadingZeros64
0.46	qemu-x86_64	[.] float64_sub
0.38	qemu-x86_64	[.] cpu_x86_exec
0.38	qemu-x86_64	[.] __pthread_mutex_unlock
0.37	qemu-x86_64	[.] float64_compare_quiet
0.32	qemu-x86_64	[.] __pthread_mutex_lock
0.25	perf_2.6.32-46	[.] gettimeofday
0.24	perf_2.6.32-46	[k] do_poll

Table 26: Profiling of 410.bwaves

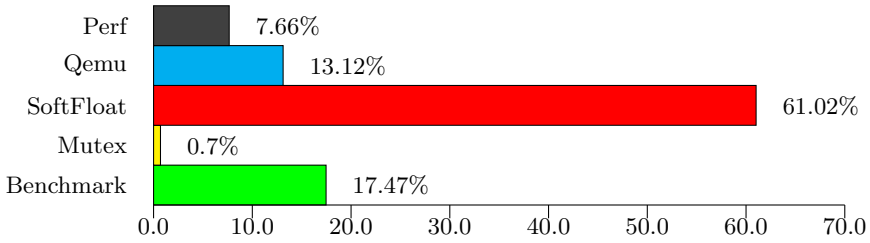

Figure 29: Main Code Areas for 410.bwaves

Overhead(%)	Command	Symbol
26.52	qemu-x86_64	[.] float64_mul
20.28	qemu-x86_64	[.] 0x007fa17b225d09
16.99	qemu-x86_64	[.] roundAndPackFloat64
7.00	qemu-x86_64	[.] addFloat64Sigs
5.05	qemu-x86_64	[.] helper_mulsd
3.42	qemu-x86_64	[.] subFloat64Sigs
3.17	qemu-x86_64	[.] float64_add
1.95	qemu-x86_64	[.] helper_addsd
1.50	qemu-x86_64	[.] normalizeRoundAndPackFloat64
1.19	qemu-x86_64	[.] cpu_x86_exec
1.13	qemu-x86_64	[.] __pthread_mutex_unlock
0.94	qemu-x86_64	[.] __pthread_mutex_lock
0.85	qemu-x86_64	[.] estimateDiv128To64
0.53	qemu-x86_64	[.] 0x0000000000c676
0.44	qemu-x86_64	[.] helper_cc_compute_all
0.37	qemu-x86_64	[.] float64_sub
0.33	qemu-x86_64	[.] float64_compare_quiet
0.28	qemu-x86_64	[.] gettimeofday
0.28	qemu-x86_64	[.] countLeadingZeros64
0.23	qemu-x86_64	[k] generic_getxattr

Table 27: Profiling of 416.gamess

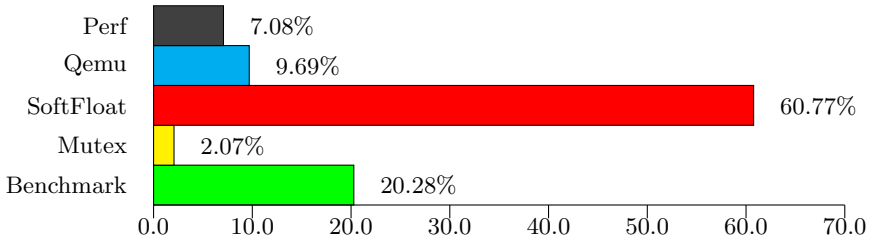

Figure 30: Main Code Areas for 416.gamess

Overhead(%)	Command	Symbol
26.23	qemu-x86_64	[.] float64_mul
23.19	qemu-x86_64	[.] roundAndPackFloat64
13.23	qemu-x86_64	[.] 0x007f4f721a9c5d
7.61	qemu-x86_64	[.] addFloat64Sigs
6.26	qemu-x86_64	[.] subFloat64Sigs
3.46	qemu-x86_64	[.] float64_add
3.15	qemu-x86_64	[.] helper_mulsd
2.84	qemu-x86_64	[.] normalizeRoundAndPackFloat64
1.36	qemu-x86_64	[.] float64_sub
1.36	qemu-x86_64	[.] __pthread_mutex_unlock
1.35	qemu-x86_64	[.] helper_addsd
1.35	qemu-x86_64	[.] cpu_x86_exec
1.32	qemu-x86_64	[.] __pthread_mutex_lock
0.51	qemu-x86_64	[.] helper_subsd
0.49	qemu-x86_64	[.] countLeadingZeros64
0.34	perf_2.6.32-46	[.] 0x0000000000c325
0.24	qemu-x86_64	[.] helper_pxor_xmm
0.21	perf_2.6.32-46	[k] schedule
0.20	perf_2.6.32-46	[k] poll_schedule_timeout
0.20	perf_2.6.32-46	[k] native_write_msr_safe

Table 28: Profiling of 433.milc

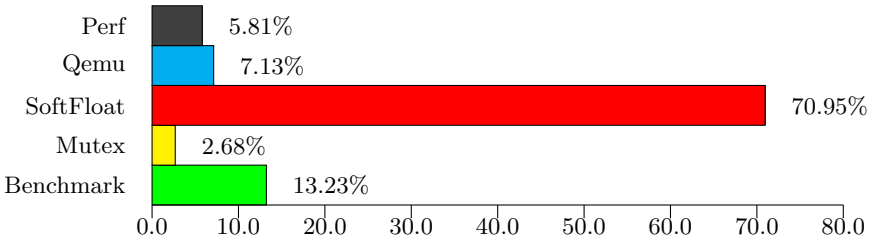

Figure 31: Main Code Areas for 433.milc

Overhead(%)	Command	Symbol
21.88	qemu-x86_64	[.] float64_mul
15.51	qemu-x86_64	[.] 0x007f086d120cd9
15.22	qemu-x86_64	[.] roundAndPackFloat64
6.52	qemu-x86_64	[.] addFloat64Sigs
6.34	qemu-x86_64	[.] estimateDiv128To64
4.64	qemu-x86_64	[.] subFloat64Sigs
3.03	qemu-x86_64	[.] helper_mulsd
2.60	qemu-x86_64	[.] float64_div
1.91	qemu-x86_64	[.] float64_compare_quiet
1.87	qemu-x86_64	[.] helper_mulpd
1.67	qemu-x86_64	[.] float64_add
1.65	qemu-x86_64	[.] float64_sub
1.47	qemu-x86_64	[.] normalizeRoundAndPackFloat64
1.40	qemu-x86_64	[.] helper_ucomisd
1.06	qemu-x86_64	[.] helper_addsd
0.94	qemu-x86_64	[.] helper_subsd
0.77	qemu-x86_64	[.] cpu_x86_exec
0.77	qemu-x86_64	[.] __pthread_mutex_unlock
0.76	qemu-x86_64	[.] helper_cc_compute_all
0.75	qemu-x86_64	[.] __pthread_mutex_lock

Table 29: Profiling of 434.zeusmp

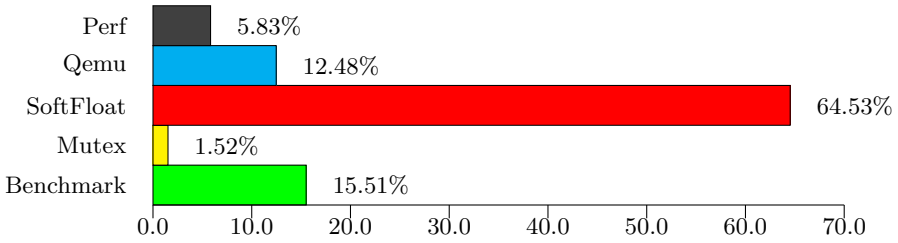

Figure 32: Main Code Areas for 434.zeusmp

Overhead(%)	Command	Symbol
24.66	qemu-x86_64	[.] roundAndPackFloat32
17.31	qemu-x86_64	[.] float32_mul
9.85	qemu-x86_64	[.] 0x0000000083a2c1
8.24	qemu-x86_64	[.] subFloat32Sigs
7.44	qemu-x86_64	[.] addFloat32Sigs
3.33	qemu-x86_64	[.] estimateSqrt32
3.12	qemu-x86_64	[.] helper_mulss
3.10	qemu-x86_64	[.] normalizeRoundAndPackFloat32
2.65	qemu-x86_64	[.] float32_div
2.08	qemu-x86_64	[.] float32_sub
1.95	qemu-x86_64	[.] float32_add
1.09	qemu-x86_64	[.] helper_addss
1.04	qemu-x86_64	[.] helper_subss
1.04	qemu-x86_64	[.] float32_sqrt
0.86	qemu-x86_64	[.] __pthread_mutex_unlock
0.85	qemu-x86_64	[.] cpu_x86_exec
0.73	qemu-x86_64	[.] __pthread_mutex_lock
0.54	perf_2.6.32-46	[.] 0x0000000000c652
0.37	qemu-x86_64	[.] float32_compare_quiet
0.27	qemu-x86_64	[.] helper_sqrtss

Table 30: Profiling of 435.gromacs

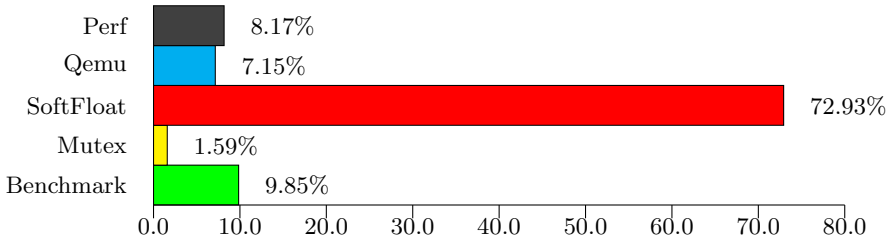

Figure 33: Main Code Areas for 435.gromacs

Overhead(%)	Command	Symbol
21.44	qemu-x86_64	[.] roundAndPackFloat64
21.34	qemu-x86_64	[.] float64_mul
10.49	qemu-x86_64	[.] addFloat64Sigs
9.28	qemu-x86_64	[.] 0x00000000856d94
8.83	qemu-x86_64	[.] subFloat64Sigs
3.83	qemu-x86_64	[.] helper_mulpd
3.50	qemu-x86_64	[.] float64_add
3.06	qemu-x86_64	[.] normalizeRoundAndPackFloat64
2.21	qemu-x86_64	[.] helper_addpd
2.19	qemu-x86_64	[.] float64_sub
1.41	qemu-x86_64	[.] helper_subpd
0.92	qemu-x86_64	[.] cpu_x86_exec
0.75	qemu-x86_64	[.] __pthread_mutex_unlock
0.71	qemu-x86_64	[.] __pthread_mutex_lock
0.56	qemu-x86_64	[.] countLeadingZeros64
0.54	perf_2.6.32-46	[.] 0x0000000000c676
0.30	perf_2.6.32-46	[k] schedule
0.30	perf_2.6.32-46	[k] poll_schedule_timeout
0.29	perf_2.6.32-46	[k] native_write_msr_safe
0.27	perf_2.6.32-46	[.] gettimeofday

Table 31: Profiling of 436.cactusADM

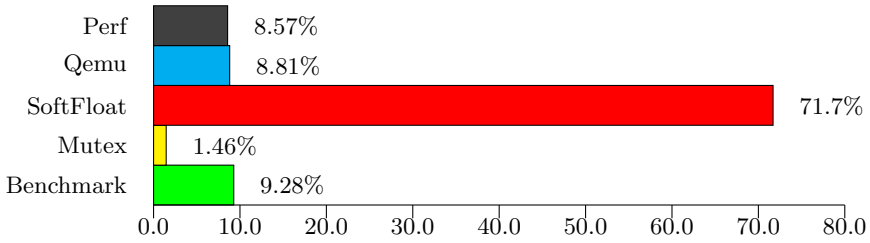

Figure 34: Main Code Areas for 436.cactusADM

Overhead(%)	Command	Symbol
23.76	qemu-x86_64	[.] float64_mul
18.03	qemu-x86_64	[.] roundAndPackFloat64
12.43	qemu-x86_64	[.] 0x0000000083b604
7.63	qemu-x86_64	[.] estimateDiv128To64
5.84	qemu-x86_64	[.] subFloat64Sigs
4.34	qemu-x86_64	[.] addFloat64Sigs
3.65	qemu-x86_64	[.] helper_mulpd
2.70	qemu-x86_64	[.] normalizeRoundAndPackFloat64
1.57	qemu-x86_64	[.] float64_add
1.45	qemu-x86_64	[.] float64_sub
1.43	qemu-x86_64	[.] float64_div
1.27	qemu-x86_64	[.] helper_subpd
1.24	qemu-x86_64	[.] helper_addpd
0.85	perf_2.6.32-46	[k] find_busiest_group
0.58	qemu-x86_64	[.] estimateSqrt32
0.57	qemu-x86_64	[.] countLeadingZeros64
0.52	perf_2.6.32-46	[.] 0x0000000000c670
0.51	qemu-x86_64	[.] cpu_x86_exec
0.47	qemu-x86_64	[.] __pthread_mutex_lock
0.42	qemu-x86_64	[.] __pthread_mutex_unlock

Table 32: Profiling of 437.leslie3d

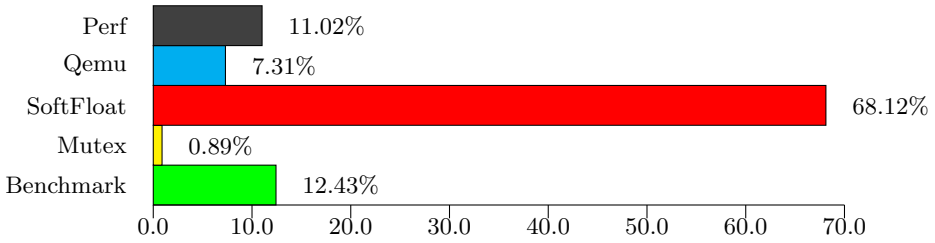

Figure 35: Main Code Areas for 437.leslie3d

Overhead(%)	Command	Symbol
25.27	qemu-x86_64	[.] roundAndPackFloat64
25.23	qemu-x86_64	[.] float64_mul
8.54	qemu-x86_64	[.] 0x000000008580d4
7.38	qemu-x86_64	[.] subFloat64Sigs
6.67	qemu-x86_64	[.] addFloat64Sigs
4.34	qemu-x86_64	[.] normalizeRoundAndPackFloat64
2.91	qemu-x86_64	[.] helper_mulsd
2.40	qemu-x86_64	[.] float64_add
1.41	qemu-x86_64	[.] float64_sub
1.32	qemu-x86_64	[.] helper_addsd
0.97	qemu-x86_64	[.] helper_subsd
0.90	qemu-x86_64	[.] float64_compare_quiet
0.79	qemu-x86_64	[.] countLeadingZeros64
0.61	qemu-x86_64	[.] helper_ucomisd
0.56	qemu-x86_64	[.] cpu_x86_exec
0.53	qemu-x86_64	[.] __pthread_mutex_unlock
0.51	perf_2.6.32-46	[.] 0x0000000000bcc3
0.44	qemu-x86_64	[.] __pthread_mutex_lock
0.44	qemu-x86_64	[.] roundAndPackFloat32
0.32	qemu-x86_64	[.] float32_to_float64

Table 33: Profiling of 444.namd

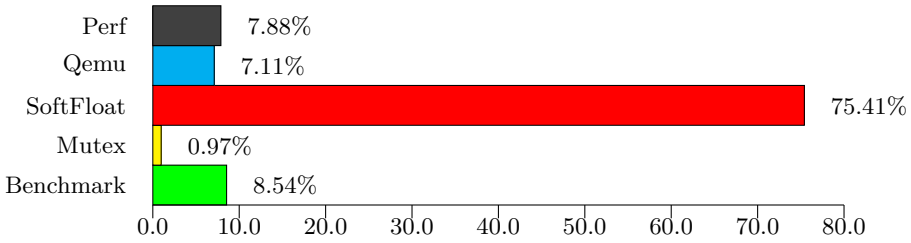

Figure 36: Main Code Areas for 444.namd

Overhead(%)	Command	Symbol
26.70	qemu-x86_64	[.] 0x000000008409f0
16.29	qemu-x86_64	[.] float64_mul
13.10	qemu-x86_64	[.] roundAndPackFloat64
8.05	qemu-x86_64	[.] __pthread_mutex_unlock
7.46	qemu-x86_64	[.] cpu_x86_exec
6.67	qemu-x86_64	[.] __pthread_mutex_lock
5.15	qemu-x86_64	[.] addFloat64Sigs
4.08	qemu-x86_64	[.] subFloat64Sigs
3.56	qemu-x86_64	[.] float64_add
2.57	qemu-x86_64	[.] helper_mulsd
2.01	qemu-x86_64	[.] normalizeRoundAndPackFloat64
1.43	qemu-x86_64	[.] helper_addsd
0.32	qemu-x86_64	[.] countLeadingZeros64
0.22	qemu-x86_64	[.] estimateDiv128To64
0.15	qemu-x86_64	[.] helper_pcmpeqb_xmm
0.10	perf_2.6.32-46	[.] 0x0000000000c1ac
0.06	qemu-x86_64	[.] helper_pxor_xmm
0.06	qemu-x86_64	[.] float64_sub
0.06	qemu-x86_64	[.] float64_div
0.05	qemu-x86_64	[.] helper_cc_compute_all

Table 34: Profiling of 447.dealII

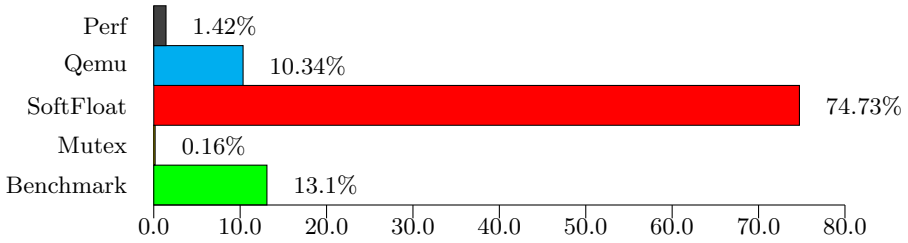

Figure 37: Main Code Areas for 447.dealII

Overhead(%)	Command	Symbol
24.68	qemu-x86_64	[.] 0x007f442b0c5d09
18.30	qemu-x86_64	[.] float64_mul
16.16	qemu-x86_64	[.] roundAndPackFloat64
5.35	qemu-x86_64	[.] addFloat64Sigs
4.88	qemu-x86_64	[.] subFloat64Sigs
2.97	qemu-x86_64	[.] normalizeRoundAndPackFloat64
2.89	qemu-x86_64	[.] __pthread_mutex_unlock
2.82	qemu-x86_64	[.] float64_compare_quiet
2.80	qemu-x86_64	[.] float64_add
2.77	qemu-x86_64	[.] helper_mulsd
2.72	qemu-x86_64	[.] estimateDiv128To64
2.40	qemu-x86_64	[.] cpu_x86_exec
2.38	qemu-x86_64	[.] __pthread_mutex_lock
1.69	qemu-x86_64	[.] helper_ucomisd
1.51	qemu-x86_64	[.] helper_addsd
1.28	qemu-x86_64	[.] helper_cc_compute_all
0.98	qemu-x86_64	[.] helper_pxor_xmm
0.59	qemu-x86_64	[.] float64_div
0.53	qemu-x86_64	[.] helper_pcmpeqb_xmm
0.49	qemu-x86_64	[.] countLeadingZeros64

Table 35: Profiling of 450.soplex

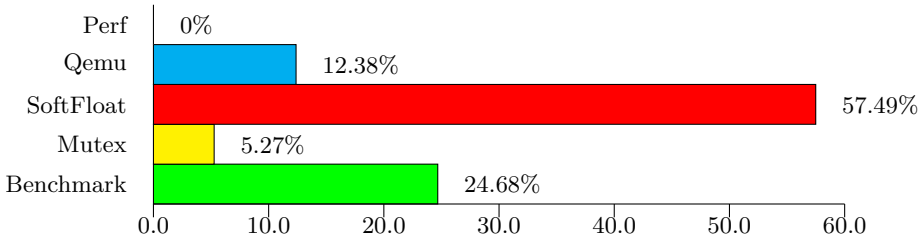

Figure 38: Main Code Areas for 450.soplex

Overhead(%)	Command	Symbol
19.89	qemu-x86_64	[.] 0x007fe29b8ead0d
14.93	qemu-x86_64	[.] roundAndPackFloat64
14.92	qemu-x86_64	[.] float64_mul
7.13	qemu-x86_64	[.] __pthread_mutex_lock
7.10	qemu-x86_64	[.] __pthread_mutex_unlock
6.36	qemu-x86_64	[.] cpu_x86_exec
5.32	qemu-x86_64	[.] addFloat64Sigs
4.96	qemu-x86_64	[.] subFloat64Sigs
2.42	qemu-x86_64	[.] normalizeRoundAndPackFloat64
2.33	qemu-x86_64	[.] float64_add
2.33	qemu-x86_64	[.] helper_mulsd
1.98	qemu-x86_64	[.] float64_compare_quiet
1.42	qemu-x86_64	[.] estimateDiv128To64
1.31	qemu-x86_64	[.] helper_addsd
1.19	qemu-x86_64	[.] helper_ucomisd
0.61	qemu-x86_64	[.] float32_to_float64
0.61	qemu-x86_64	[.] helper_cc_compute_all
0.42	qemu-x86_64	[.] float64_sub
0.41	qemu-x86_64	[.] countLeadingZeros64
0.39	qemu-x86_64	[.] helper_cvtps2pd

Table 36: Profiling of 453.povray

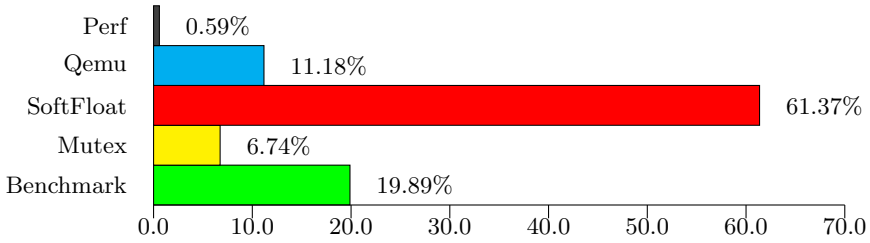

Figure 39: Main Code Areas for 453.povray

Overhead(%)	Command	Symbol
37.46	qemu-x86_64	[.] float64_mul
23.21	qemu-x86_64	[.] roundAndPackFloat64
5.90	qemu-x86_64	[.] 0x000000028850a0
4.80	qemu-x86_64	[.] addFloat64Sigs
4.61	qemu-x86_64	[.] helper_mulsd
3.90	qemu-x86_64	[.] subFloat64Sigs
3.71	qemu-x86_64	[.] float64_add
2.10	qemu-x86_64	[.] normalizeRoundAndPackFloat64
1.35	qemu-x86_64	[.] helper_addsd
0.53	perf_2.6.32-46	[.] 0x0000000000bcc3
0.52	qemu-x86_64	[.] cpu_x86_exec
0.45	qemu-x86_64	[.] __pthread_mutex_unlock
0.37	qemu-x86_64	[.] __pthread_mutex_lock
0.36	qemu-x86_64	[.] countLeadingZeros64
0.36	perf_2.6.32-46	[k] do_poll
0.34	perf_2.6.32-46	[k] schedule
0.32	perf_2.6.32-46	[k] generic_getxattr
0.28	perf_2.6.32-46	[k] native_write_msr_safe
0.28	perf_2.6.32-46	[.] gettimeofday
0.27	perf_2.6.32-46	[k] find_busiest_group

Table 37: Profiling of 454.calculix

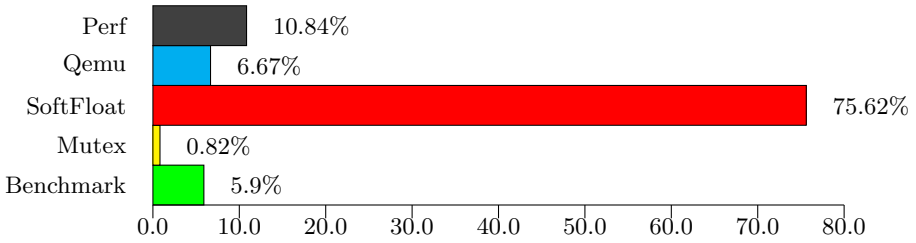

Figure 40: Main Code Areas for 454.calculix

Overhead(%)	Command	Symbol
24.77	qemu-x86_64	[.] float64_mul
24.37	qemu-x86_64	[.] roundAndPackFloat64
12.13	qemu-x86_64	[.] 0x0000000083d964
8.31	qemu-x86_64	[.] subFloat64Sigs
4.98	qemu-x86_64	[.] addFloat64Sigs
4.83	qemu-x86_64	[.] normalizeRoundAndPackFloat64
2.96	qemu-x86_64	[.] helper_mulpd
2.08	qemu-x86_64	[.] float64_add
1.64	qemu-x86_64	[.] float64_sub
1.39	qemu-x86_64	[.] helper_addpd
1.29	qemu-x86_64	[.] helper_subpd
0.92	qemu-x86_64	[.] countLeadingZeros64
0.83	qemu-x86_64	[.] __pthread_mutex_unlock
0.80	qemu-x86_64	[.] cpu_x86_exec
0.69	qemu-x86_64	[.] __pthread_mutex_lock
0.48	perf_2.6.32-46	[.] 0x0000000000c0e6
0.27	qemu-x86_64	[.] helper_mulsd
0.25	perf_2.6.32-46	[k] generic_getxattr
0.24	perf_2.6.32-46	[.] gettimeofday
0.21	qemu-x86_64	[.] helper_imulq_T0_T1

Table 38: Profiling of 459.GemsFDTD

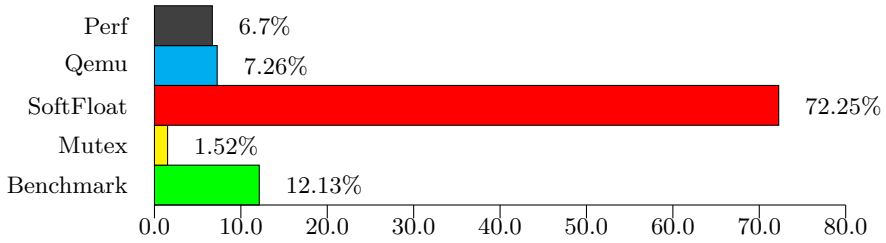

Figure 41: Main Code Areas for 459.GemsFDTD

Overhead(%)	Command	Symbol
22.33	qemu-x86_64	[.] float64_mul
18.37	qemu-x86_64	[.] roundAndPackFloat64
17.16	qemu-x86_64	[.] 0x007fbeec23ecd9
5.47	qemu-x86_64	[.] addFloat64Sigs
3.82	qemu-x86_64	[.] subFloat64Sigs
3.24	qemu-x86_64	[.] helper_mulsd
2.32	qemu-x86_64	[.] cpu_x86_exec
2.28	qemu-x86_64	[.] normalizeRoundAndPackFloat64
2.17	qemu-x86_64	[.] float64_add
2.13	qemu-x86_64	[.] __pthread_mutex_unlock
1.79	qemu-x86_64	[.] __pthread_mutex_lock
1.35	qemu-x86_64	[.] estimateDiv128To64
1.34	qemu-x86_64	[.] helper_addsd
0.59	perf_2.6.32-46	[k] find_busiest_group
0.56	qemu-x86_64	[.] float64_sub
0.51	qemu-x86_64	[.] sincos
0.48	perf_2.6.32-46	[.] 0x0000000000c670
0.45	qemu-x86_64	[.] helper_imulq_T0_T1
0.42	qemu-x86_64	[.] countLeadingZeros64
0.38	qemu-x86_64	[.] helper_subsd

Table 39: Profiling of 465.tonto

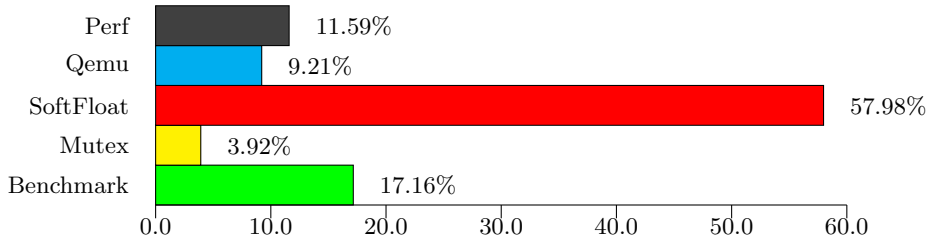

Figure 42: Main Code Areas for 465.tonto

Overhead(%)	Command	Symbol
27.34	qemu-x86_64	[.] roundAndPackFloat64
20.59	qemu-x86_64	[.] float64_mul
9.95	qemu-x86_64	[.] subFloat64Sigs
6.53	qemu-x86_64	[.] 0x0000000084108f
6.42	qemu-x86_64	[.] addFloat64Sigs
5.71	qemu-x86_64	[.] normalizeRoundAndPackFloat64
2.84	qemu-x86_64	[.] helper_mulsd
2.54	qemu-x86_64	[.] float64_add
2.22	qemu-x86_64	[.] estimateDiv128To64
2.11	qemu-x86_64	[.] helper_addsd
1.43	qemu-x86_64	[.] float64_sub
1.12	qemu-x86_64	[.] countLeadingZeros64
1.09	qemu-x86_64	[.] helper_subsd
0.81	qemu-x86_64	[.] __pthread_mutex_unlock
0.78	qemu-x86_64	[.] __pthread_mutex_lock
0.75	qemu-x86_64	[.] cpu_x86_exec
0.53	perf_2.6.32-46	[.] 0x0000000000c0e6
0.49	qemu-x86_64	[.] float64_div
0.27	perf_2.6.32-46	[.] gettimeofday
0.25	perf_2.6.32-46	[k] generic_getxattr

Table 40: Profiling of 470.lbm

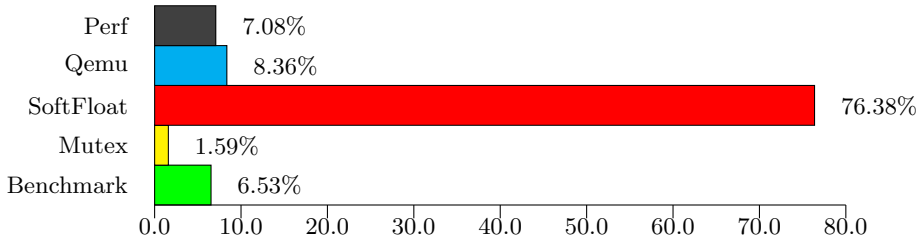

Figure 43: Main Code Areas for 470.lbm

Overhead(%)	Command	Symbol
26.44	qemu-x86_64	[.] roundAndPackFloat32
19.72	qemu-x86_64	[.] float32_mul
9.54	qemu-x86_64	[.] 0x0000000083ce8c
7.82	qemu-x86_64	[.] addFloat32Sigs
7.25	qemu-x86_64	[.] subFloat32Sigs
3.36	qemu-x86_64	[.] float32_div
3.00	qemu-x86_64	[.] helper_mulps
2.81	qemu-x86_64	[.] normalizeRoundAndPackFloat32
2.48	qemu-x86_64	[.] float32_add
2.05	qemu-x86_64	[.] float32_sub
1.07	qemu-x86_64	[.] helper_mulss
0.94	qemu-x86_64	[.] helper_addps
0.93	qemu-x86_64	[.] helper_subps
0.84	qemu-x86_64	[.] cpu_x86_exec
0.79	qemu-x86_64	[.] __pthread_mutex_unlock
0.67	qemu-x86_64	[.] __pthread_mutex_lock
0.54	perf_2.6.32-46	[.] 0x0000000000c20f
0.50	qemu-x86_64	[.] helper_addss
0.35	qemu-x86_64	[.] helper_subss
0.32	perf_2.6.32-46	[k] generic_getxattr

Table 41: Profiling of 481.wrf

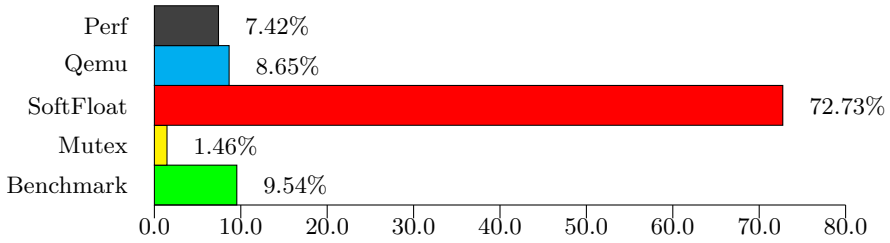

Figure 44: Main Code Areas for 481.wrf

Overhead(%)	Command	Symbol
21.02	qemu-x86_64	[.] float64_mul
14.09	qemu-x86_64	[.] roundAndPackFloat64
10.85	qemu-x86_64	[.] 0x007fd217bc8cd9
6.79	qemu-x86_64	[.] float32_to_float64
5.27	qemu-x86_64	[.] roundAndPackFloat32
5.07	qemu-x86_64	[.] helper_punpckldq_xmm
4.54	qemu-x86_64	[.] helper_cvtps2pd
4.02	qemu-x86_64	[.] addFloat32Sigs
3.46	qemu-x86_64	[.] addFloat64Sigs
3.00	qemu-x86_64	[.] subFloat32Sigs
2.25	qemu-x86_64	[.] helper_mulsd
2.10	qemu-x86_64	[.] float32_sub
1.88	qemu-x86_64	[.] subFloat64Sigs
1.33	qemu-x86_64	[.] float64_sub
1.09	qemu-x86_64	[.] normalizeRoundAndPackFloat64
0.95	qemu-x86_64	[.] normalizeRoundAndPackFloat32
0.80	qemu-x86_64	[.] helper_subss
0.71	qemu-x86_64	[.] helper_subsd
0.59	perf_2.6.32-46	[.] 0x0000000000c670
0.49	qemu-x86_64	[.] __pthread_mutex_unlock

Table 42: Profiling of 482.sphinx3

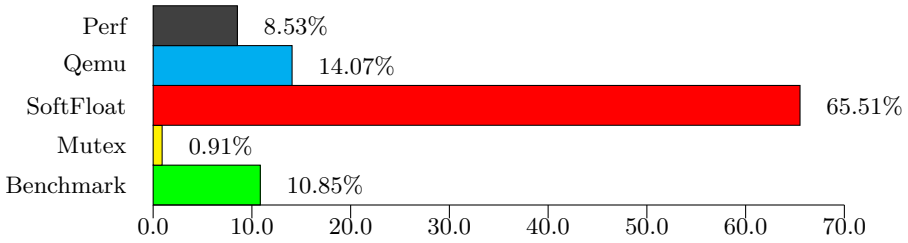

Figure 45: Main Code Areas for 482.sphinx3