

# Performance Characterization of SPEC CPU2006 Integer Benchmarks on x86-64 Architecture

Dong Ye\*, Joydeep Ray<sup>†</sup>, Christophe Harle<sup>†</sup>, and David Kaeli\*

\*ECE Department, Northeastern University, Boston, MA 02115

Email: dye, kaeli@ece.neu.edu

<sup>†</sup>Advanced Micro Devices, Austin, TX 78741

Email: joydeep.ray, christophe.harle@amd.com

**Abstract**—As x86-64 processors become the CPU of choice for the personal computer market, it becomes increasingly important to understand the performance we can expect by migrating applications from a 32-bit environment to a 64-bit environment. For applications that can effectively exploit a larger memory address space (e.g., commercial databases and digital content authoring tools), it is not surprising that x86-64 can provide a performance boost. However, for less-demanding desktop applications that can fit in a 32-bit address space, we would like to know if we can expect any performance benefits by moving to this platform.

In this paper, we report on a range of performance characteristics for programs compiled for both 32 bits and 64 bits and run directly (32-bit binaries are run in compatibility mode<sup>1</sup>; 64-bit binaries are run in 64-bit mode) on a single x86-64 based system. In this study we utilize the integer benchmarks from the newly released SPEC CPU2006 suite.

We have observed that for the SPEC CPU2006 integer benchmarks, 64-bit mode offers a sizable performance advantage over 32-bit mode (7% on average). However, the advantages vary from benchmark to benchmark, and for a handful of programs, 64-bit mode is significantly slower than 32-bit mode (in this subset of benchmarks, performance is reduced by more than 16% when running in 64-bit mode.) We further analyze 5 benchmarks that exhibit significant differences in performance between these two modes. For this set of CPU2006 integer programs, we present a range of performance characteristics that illustrate the impact of moving to a 64-bit environment. Our results and analysis can be used by performance engineers and developers to better understand how to exploit the capabilities of the x86-64 architecture.

## I. INTRODUCTION

AMD x86-64 technology builds on top of the legacy 32-bit x86 architecture and provides backward compatibility for the existing x86 code base. It provides a number of new features including 64-bit addressability, 8 new 64-bit general-purpose registers and 8 new 128-bit SSE registers [1], as well as 64-bit extension of the existing general-purpose registers. The change of programming model embodied in these extensions helps increase the performance of applications that need to address a large amount of memory or that are register constrained [2]. Some notable examples of such applications include server applications (e.g., commercial databases) and professional multimedia content authoring applications.

Meanwhile, there has been a growing demand for 64-bit capability in the consumer computing market to overcome the

4GB addressing limit, as experienced on 32-bit x86 architectures [3]. In practice, the majority of user-level applications are limited by the 2GB virtual address space in mainstream 32-bit consumer operating systems, unless some hand-tuned hardware or operating system tricks are employed [4].

Since a single x86-64 platform can provide an environment to run both legacy 32-bit x86 code and new 64-bit code directly, we would like to investigate the potential benefits of making this move. Our goal will be to study a set of popular applications in both modes. One outcome will be to identify some program characteristics that benefit or don't benefit the performance when making the move to 64 bits. We mostly target the integer benchmarks taken from the latest SPEC CPU suite, CPU2006 [5], in this study. We also revisit the integer benchmarks in the SPEC CPU2000 [12] suite and investigate the impact of moving to a 64-bit environment on this suite as well as the difference between these two suites.

As a number of x86-64 based 64-bit processors have recently been introduced, we want to understand if 64-bit mode provides significant performance advantages, and where. From this analysis, we hope to learn how to take better advantage of the benefits available in 64-bit mode (e.g., more general-purpose registers and native 64-bit arithmetic support), as well as to what extent some concerns (e.g., larger code size and larger data size) impact these workloads.

Figure 1 presents the speedup obtained running the CPU2006 integer benchmarks in 64-bit mode versus in 32-bit mode, both run with the reference inputs in the suite. The programs were run on an AMD Athlon™ 64 processor, which is based the AMD-K8™ microarchitecture (an implementation of x86-64 architecture). All programs were compiled with the GCC 4.1.1 C/C++ compiler [8]. As we can see, 64-bit mode provides a 7% performance boost over 32-bit mode on average.

In Table I we list five benchmarks from this suite that exhibited significant performance differences between these two modes; three of them performed better in 64-bit mode and two of them performed better in 32-bit mode. This set provides a good medium for discussion of the trade-offs associated with moving from a 32-bit to a 64-bit environment.

The remainder of this paper is organized as follows. Section II describes the experimental system that was used for this study. Section III describes the performance characteristics obtained during the execution of each benchmark. Section IV

<sup>1</sup>In this paper we will call this mode 32-bit mode to emphasize the 32-bit vs. 64-bit comparison.

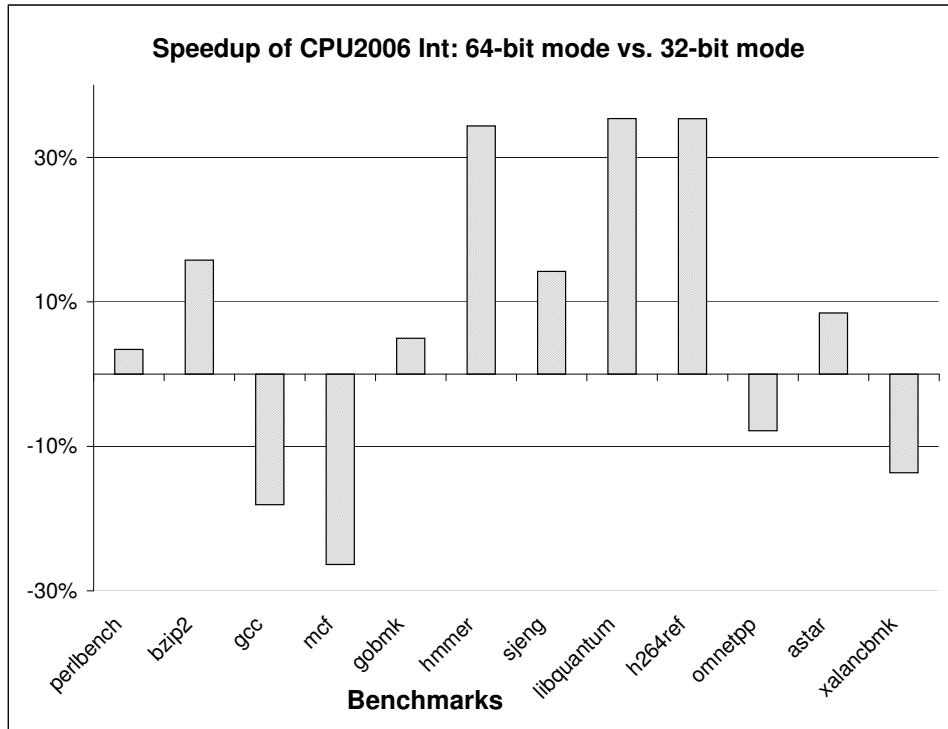


Fig. 1. Speedup of SPEC CPU2006 integer benchmarks running in 64-bit mode vs. in 32-bit mode of x86-64 architecture

presents a performance analysis using the five benchmarks in Table I that showed significant performance difference between these two modes. We also identify some typical program characteristics that favor 64-bit mode over 32-bit mode and vice versa. Section V summarizes our comparison and gives some suggestions about how better to take advantage of the capabilities of x86-64 architecture. Section V also briefly describes the performance difference between these two modes for the previous generation of the SPEC CPU suite and compares these two suites against each other. Section VI concludes the paper.

Benchmark	Speedup of 64-bit mode over 32-bit mode
mcf	-26.35%
hmmer	34.34%
libquantum	35.38%
h264ref	35.35%
xalancbmk	-13.65%

TABLE I

FIVE SPEC CPU2006 INTEGER BENCHMARKS SHOWED SIGNIFICANT PERFORMANCE DIFFERENCE BETWEEN 64-BIT MODE AND 32-BIT MODE.

## II. EXPERIMENTAL SYSTEM SETUP

Our characterization study was performed on an AMD Athlon™ 64 X2 4400+ dual-core microprocessor. The system was populated with 2x1GB DDR400 memory chips (two PC3200 DIMMs). The AMD Athlon™ 64 X2 microprocessor integrates two cores on the single chip. The two cores

share a single on-chip memory controller as a memory interface to the on-board memory chips, and a single on-chip HyperTransport™ link as an I/O interface to the on-board I/O bus. Both cores were clocked at 2.2 GHz. The on-chip memory controller provides a dual-channel 128-bit wide interface to main memory. In this system setting the memory interface gave a peak memory bandwidth of 6.4 GB/sec.

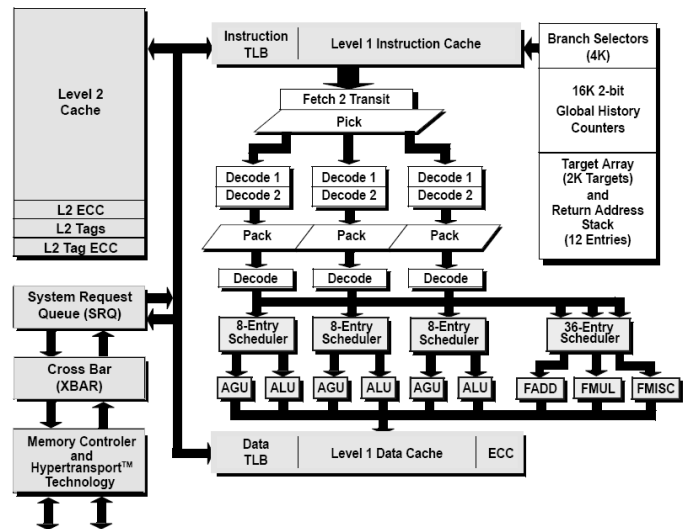


Fig. 2. A block diagram of the AMD Athlon 64 processor

Figure 2 shows a block diagram of an AMD Athlon™ 64 single-core chip, which is essentially duplicated on the dual-

core chip used in this system. The exceptions are the crossbar, the memory controller and the HyperTransport™ link. These units are all shared between the two cores in the dual-core chip. The memory interface in the single-core chip is 64 bits wide while it is 128 bits wide in the dual-core chip.

Each CPU core has the following major components [6]: (1) separate L1 instruction and data caches - each 64 KB, 2-way associative with a 64 byte line size; (2) unified 1 MB L2 cache sized, 16-way set associative with a 64 byte line size; (3) separate fully-associative L1 instruction and L1 data TLBs, 32-entries each (each entry indexes to a 4 KB page); (4) separate 4-way set-associative L2 instruction and L2 data TLBs, 512 entries each (each entry indexes to a 4 KB page); (5) 16K-entry 2-bit global history branch counters; (6) 2K-entry branch target buffer; (7) 12-entry return address stack.

The major features of the pipeline includes: (1) a 3-wide instruction decoder; (2) 72-entry reorder buffer; (3) 44-entry load/store queue; (4) 3 fully-pipelined integer execution units; each has an 8-entry reservation station; (4) 3 fully-pipelined floating-point execution units; they share a 36-entry scheduler.

The operating system used was SUSE® Linux® Professional 9.3 x86-64 edition [7]. The operating system was run in level 3 mode while all the performance data was collected. This was to minimize the overhead introduced by concurrent running processes on the system, especially removing the impact of some heavy-weight processing such as the X windowing system and GUI applications. In our experiments, we observed less than 1% run-to-run variations for each individual benchmark across the entire suite.

The kernel version of this Linux® distribution is 2.6.11.4, with SMP support enabled. In order to reduce the variation introduced by the operating system’s process scheduling over multiple cores, we ran the benchmark programs exclusively on a single core during the experiment by wrapping the benchmarking invocation command inside the operating system’s process CPU-binding command.

Both the CPU and the operating system used can support direct execution of both 64-bit x86-64 binary and 32-bit x86 binary. All of our experiments were performed on this same system (no multiple-booting).

All programs except perlbench were compiled using the “-O3”, “-ffast-math”, “-funroll-all-loops” and “-march=k8” flags to generate both 32-bit and 64-bit binaries, where 64-bit binaries were generated by adding the “-m64” mode switch flag and 32-bit binaries were generated by adding the “-m32” mode switch flag. Since perlbench in CPU2006 cannot compile successfully on our experimental system with “-O3” and “-funroll-all-loops” flags, we instead used the optimization flags “-O2”, “-ffast-math” and “-march=k8” to compile this benchmark along with the mode switch flags. The binary of the compiler itself was 64-bit. This compiler is able to generate both 64-bit binaries and 32-bit binaries, depending on which mode switch flag is used. In our experiments, we used the same compiler and the same optimization switches to generate both 64-bit and 32-bit binaries. All the performance data was collected from hardware counters available on this processor

using AMD internal tools and OProfile [9].

### III. SYSTEM-WIDE PERFORMANCE CHARACTERISTICS

In this section we characterize the system-wide (including both CPU pipeline and memory subsystem) performance of the SPEC CPU2006 integer benchmarks. Our goal is to obtain a general understanding of the performance differences between 64-bit mode and 32-bit mode. The five benchmarks that showed significant performance differences between these two modes are listed in Table I. They will be the target of our discussion in Section IV.

#### A. Static Code Size and Dynamic Instruction Count

One of the major concerns when migrating to 64-bit x86-64 computing is that the size of the generated binary increases due to the increased length of instructions (10% on average as reported in [10]), and the sizes of some data types (e.g., pointers and long integers) get doubled. The increased binary size may reduce the efficiency of the instruction cache and the increased length of instruction may introduce extra pressure on the bandwidth of the decoder. The doubling in the sizes of pointer and long data types may increase the runtime memory footprint, especially those programs that heavily utilize these data types [10].

Figure 3 shows the code (text section of the generated binary) size increase of 64-bit mode versus 32-bit mode. Figure 4 shows the increase in the size of the steady-state runtime memory footprint observed during the program lifetime from 32-bit mode to 64-bit mode. Note that both versions of binaries are dynamically linked. Therefore, in our comparison of the runtime memory footprint, the observed difference in memory footprint can be attributed to both the different application codes and the library codes executed in these two modes.

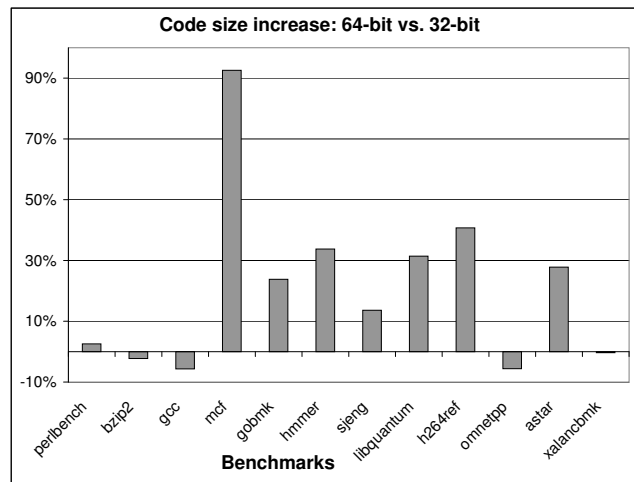


Fig. 3. 64-bit binary is larger than 32-bit binary by the amount of 21% on average for the CPU2006 integer benchmarks.

It is interesting to note that two benchmarks, bzip2 and sjeng, exhibit little if any differences in their runtime memory

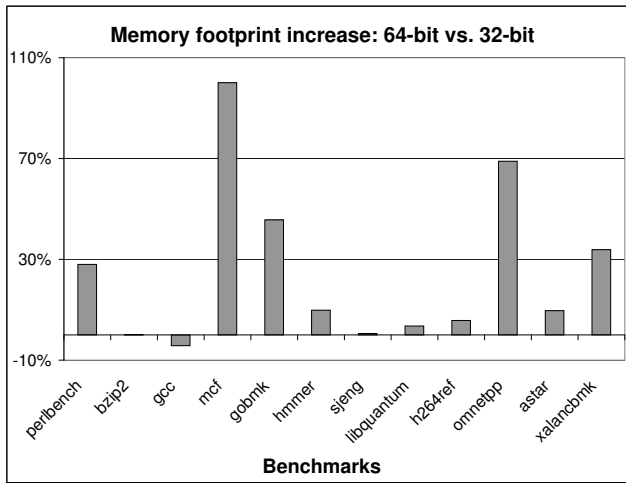


Fig. 4. 64-bit mode execution has a larger runtime memory footprint than 32-bit mode by 25.1% on average for the CPU2006 integer benchmarks.

footprints between 64-bit mode and 32-bit mode. One benchmark, gcc, even shows a decrease in the size of its runtime memory footprint in 64-bit mode.

Compared to x86, x86-64 architecture doubles the number of general-purpose and SSE registers, and doubles the width of all the general-purpose registers. The x86-64 architecture also provides a flat 64-bit address space. All these features are enabled in 64-bit mode but not in 32-bit mode. We expect that these features will increase code density. Figure 5 shows the decrease in the dynamic instruction count in 64-bit mode, which confirms our expectation. On average, the dynamic instruction count is decreased by 12% across the CPU2006 integer benchmarks.

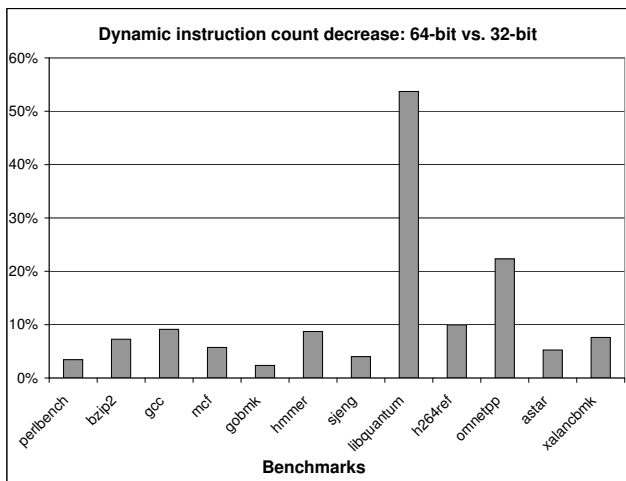


Fig. 5. The number of instructions dynamically executed is 12% less in 64-bit mode than in 32-bit mode.

### B. Comparison of CPU Utilization

Figure 6 shows the instructions-per-cycle (IPC) for the CPU2006 integer benchmarks run in 64-bit mode versus 32-

bit mode. We have already showed the impact on the number of dynamically executed instructions in Figure 5. From these figures, we can get a first-order understanding about which component(s) is responsible for the observed performance difference. This will guide us in the investigation of individual benchmarks in Section IV.

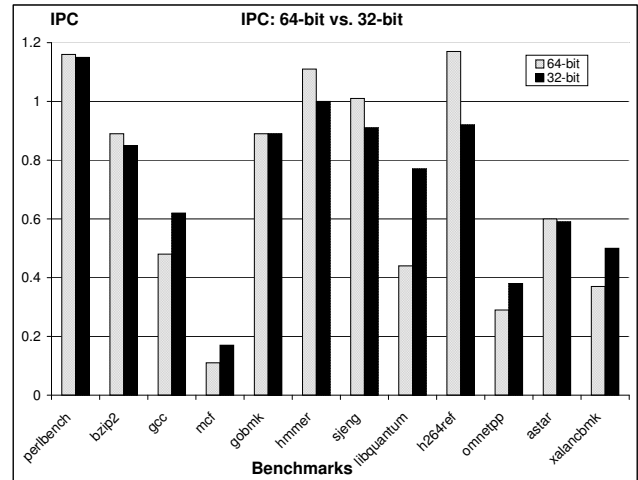


Fig. 6. The IPC observed in 64-bit mode decreases by 7.8% on average versus that in 32-bit mode across the CPU2006 integer benchmarks.

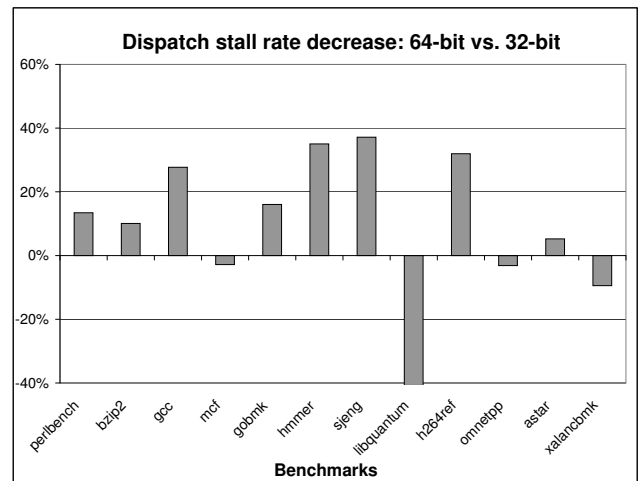


Fig. 7. The dispatch stall rate (dispatch stall cycles per one thousand retired instructions) observed in 64-bit mode decreases by 10% on average versus that in 32-bit mode across the CPU2006 integer benchmarks.

To evaluate the increased pressure placed on instruction fetch and decode unit due to the factor of longer opcodes and the longer instructions in 64-bit mode, we compare the total decode stall cycles in both modes. The decode unit can be stalled in a number of situations: 1) when reservation stations are fully occupied, 2) when the reorder buffer is fully occupied, 3) when the load store buffer is fully occupied, 4) when fetched instructions are depleted, and 5) when branches are squashed. This comparison will give us a better understanding if these concerns are present in these applications.

We normalize decode stall cycles by dividing this number by the number of committed instructions and compare the resulting decode stall rate between the two modes. In Figure 7 we find that for the CPU2006 integer benchmarks, the anticipated adverse effect that 64-bit computing has on the efficiency of the instruction front-end is not substantiated. We find that the average dispatch stall rate across all the CPU2006 integer benchmarks decreases by 10% in 64-bit mode. The libquantum benchmark is especially interesting since it is running more than 35% faster in 64-bit mode than in 32-bit mode (as shown in Figure 1), yet it is the only benchmark in this suite that experiences a significant increase in the dispatch stall rate in 64-bit mode. We study this benchmark further in Section IV-C to attempt to explain this phenomenon by inspecting various program characteristics in both 32-bit and 64-bit modes.

### C. Memory Subsystem Performance

In this section we continue to investigate differences between these two modes in terms of their demands placed on the memory subsystem, as well as how the memory subsystem responds to the added load in 64-bit mode. We can see from Figure 3 and Figure 4 that 64-bit mode and 32-bit mode present significantly different binary sizes and memory footprints in nearly half of the CPU2006 integer benchmarks.

We first look at instruction cache request rates and instruction cache miss rates to see whether we can explain the observation we made in Figure 7 that 64-bit mode (versus 32-bit mode) does not place added pressure on the pipeline front-end.

Figure 8 and Figure 9 show comparisons of the instruction cache request rate and the instruction cache miss rate in these two modes, respectively.

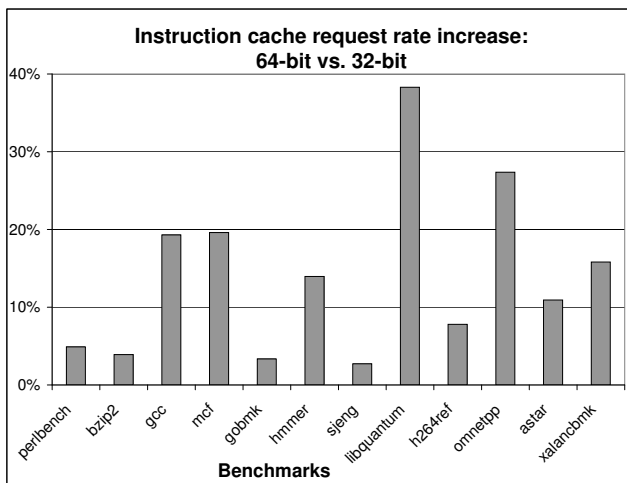


Fig. 8. The instruction cache request rate observed in 64-bit mode increases by 14% on average versus that in 32-bit mode across the CPU2006 integer benchmarks.

Although the relative difference of instruction cache miss rates between these two modes is very high (on average, an 83% increase from 32-bit mode to 64-bit mode across these benchmarks), the instruction cache miss rate is very small in

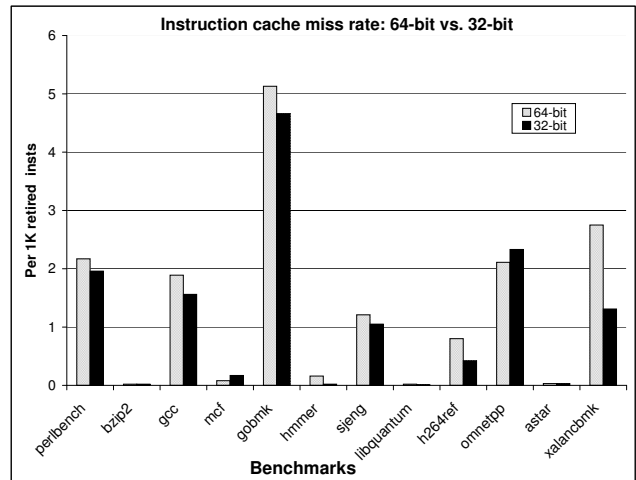


Fig. 9. The instruction cache miss rates observed in both 64-bit mode and 32-bit mode are very small for all the CPU2006 integer benchmarks. Note that the Y axis is the number of misses per 1,000 retired instructions.

both modes for all the CPU2006 integer benchmarks. This fact can help to explain the phenomenon we observed in Figure 7, which would otherwise suggest 64-bit mode had a higher dispatch stall rate because of its higher instruction cache miss rate.

Figure 10 and Figure 11 show comparisons of the data cache request rate and the data cache miss rate between these two modes, respectively.

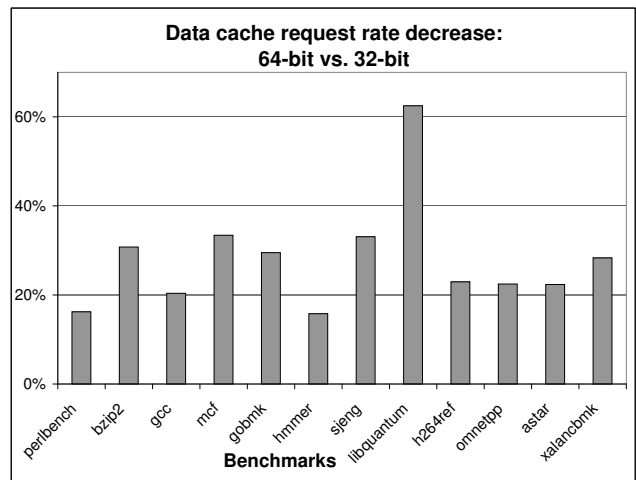


Fig. 10. The data cache request rate observed in 64-bit mode decreases by 28% on average versus that in 32-bit mode across the CPU2006 integer benchmarks.

An across-the-board decrease in the data cache request rate in 64-bit mode (as observed in Figure 10) illustrates the power of having more registers available for the compiler in 64-bit mode to reduce the number of memory accesses.

However, the significant increase in the data cache miss rate when moving from 32-bit mode to 64-bit mode indicates that doubling the size of pointer and long data types does have an adverse effect on the data cache behavior. Coupled with the

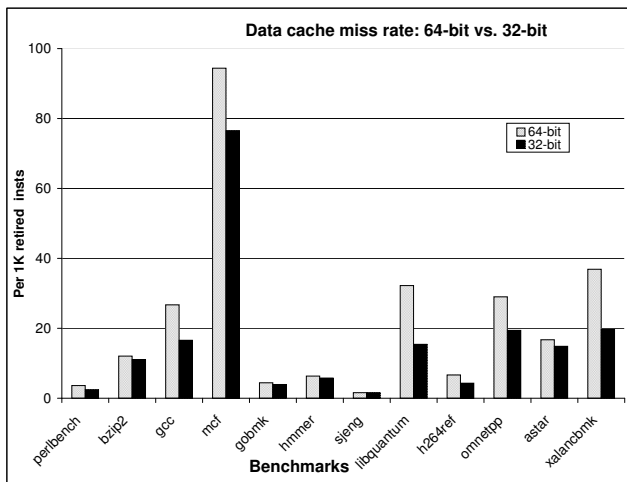


Fig. 11. The data cache miss rate observed in 64-bit mode increases significantly (nearly 40% on average) versus that in 32-bit mode. Note that the Y axis is the number of misses per 1,000 retired instructions.

fact that the absolute value of the data cache miss rates for some benchmarks (e.g., *mcf* and *xalancbmk*) is already quite high in 32-bit mode, these benchmarks take a big performance hit when moving to 64-bit mode.

Again the benchmark *libquantum* is an interesting case since the data cache miss rate in 64-bit mode is more than doubling the data cache miss rate in 32-bit mode, and both miss rates are quite significant. Even so, the performance of this benchmark is 35% better in 64-bit mode than in 32-bit mode. We will investigate the program characteristics of this benchmark in Section IV-C to better explain this phenomenon.

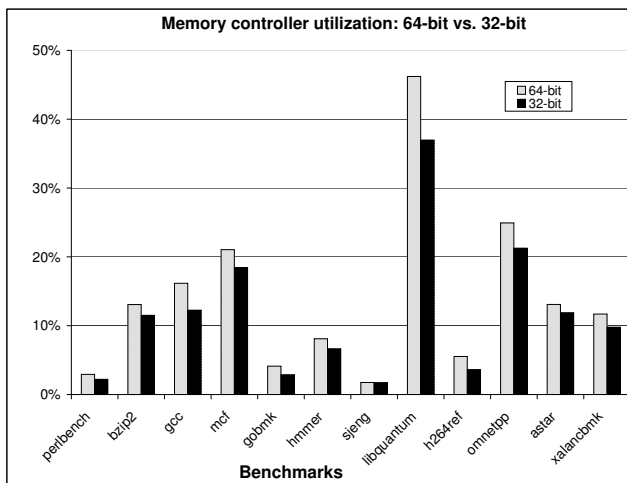


Fig. 12. The memory controller utilization observed in 64-bit mode is nearly 20% higher than that in 32-bit mode. Note that the maximum available memory controller bandwidth on the experimental system was 6.4GB/s and the memory controller utilization was calculated by dividing the observed memory bandwidth by this maximum bandwidth.

Figure 12 gives the overall average memory controller utilization experienced during the program lifetime. Comparing Figure 12 and Figure 1, we can see that applications that

have higher memory utilization are more likely to suffer performance degradation when moving to 64 bits. Again, *libquantum* is an exception to this general observation. Even if it commands the highest memory controller utilization in both 32-bit and 64-bit modes, and even if it has 25% greater utilization of the available memory controller bandwidth in 64-bit mode than in 32-bit mode, it still obtains a boost in performance by more than 35% by moving to 64 bits.

#### IV. INDIVIDUAL BENCHMARK PERFORMANCE CHARACTERIZATION

In this section we focus our analysis on five selected benchmarks from this suite: *mcf*, *hmmer*, *libquantum*, *h264ref*, and *xalancbmk*. These five benchmarks showed significant performance differences between 64-bit mode and 32-bit mode on our system, as presented in Table I. Three of them (*hmmer*, *libquantum* and *h264ref*) performed better in 64-bit mode, and two (*mcf* and *xalancbmk*) performed better in 32-bit mode.

##### A. *mcf*

*mcf* is derived from a program used for single-depot vehicle scheduling in public mass transportation. As shown in Figure 4, *mcf* has a much larger memory footprint in 64-bit mode compared to that in 32-bit mode. *mcf* has two key data structures, both of which include many elements using the long data type and the pointer data type. The sizes of long and pointer data types are doubled in 64-bit mode. As a result, these two key data structures are much bigger in 64-bit, as shown in Table II.

struct name	32-bit size	64-bit size	number of live instances at run time
node_t	56 bytes	104 bytes	50275
act.t	32 bytes	64 bytes	>27 million

TABLE II

KEY DATA STRUCTURES IN MCF ARE SIGNIFICANTLY BIGGER IN 64-BIT MODE THAN IN 32-BIT MODE.

The consequence is evident in the memory footprint, the data cache miss and the memory controller utilization shown in Figure 4, Figure 11, and Figure 12, respectively.

##### B. *hmmer*

*hmmer* is an implementation of profile HMM (Hidden Markov models) software for protein sequence analysis. In both modes, more than 97% of runtime is spent in a single function. The hottest loop in this function consumes more than half of the function's runtime (line 68 of the function `fast_algorithm.c`) Inside this hot loop, elements from 16 different arrays are accessed and therefore this loop places a great amount of pressure on register allocation. This helps to explain why 64-bit code performs much better than 32-bit code.

We can observe that 64-bit code obtains both decreased dynamic instruction count and lower IPC in Figure 5 and Figure 6, respectively.

### C. libquantum

libquantum is a library for the simulation of a quantum computer. The reason this benchmark is so much faster in 64-bit mode is because it uses 64-bit integer arithmetic intensively in its algorithm. In the 64-bit mode of x86-64 architecture, the 64-bit arithmetic operations are performed directly in hardware, whereas in 32-bit mode, no 64-bit arithmetic operations are supported directly in hardware and thus a single 64-bit arithmetic operation has to be implemented with multiple 32-bit arithmetic operations.

The effect of 64-bit arithmetic support in 64-bit mode is evident in the significant decrease of dynamic instruction count in 64-bit mode, as shown in Figure 5. The dynamic instruction count is 54% smaller in 64-bit mode than in 32-bit mode. This also explains why both the dispatch stall rate in Figure 7 and the data cache miss rate in Figure 11 seem to indicate that this benchmark will suffer a significant performance degradation in 64-bit mode, whereas the results turn out otherwise.

### D. h264ref

h264ref is a reference implementation of the H.264/AVC video compression standard [11]. In this benchmark, the loop starting from line 417 in the file of `mv-search.c` is the hottest spot in the code. The loop body is shown in Figure 13.

```
for (y = 0; y < 4; y++)
{
    refptr = PeYline_11 (ref_pic, abs_y++, abs_x, img_height, img_width);
    LineSadBlk0 += byte_abs [refptr++ - *orgptr++];
    LineSadBlk0 += byte_abs [refptr++ - *orgptr++];
    LineSadBlk0 += byte_abs [refptr++ - *orgptr++];
    LineSadBlk0 += byte_abs [refptr++ - *orgptr++];
    LineSadBlk1 += byte_abs [refptr++ - *orgptr++];
    LineSadBlk1 += byte_abs [refptr++ - *orgptr++];
    LineSadBlk1 += byte_abs [refptr++ - *orgptr++];
    LineSadBlk1 += byte_abs [refptr++ - *orgptr++];
    LineSadBlk2 += byte_abs [refptr++ - *orgptr++];
    LineSadBlk2 += byte_abs [refptr++ - *orgptr++];
    LineSadBlk2 += byte_abs [refptr++ - *orgptr++];
    LineSadBlk2 += byte_abs [refptr++ - *orgptr++];
    LineSadBlk3 += byte_abs [refptr++ - *orgptr++];
    LineSadBlk3 += byte_abs [refptr++ - *orgptr++];
    LineSadBlk3 += byte_abs [refptr++ - *orgptr++];
    LineSadBlk3 += byte_abs [refptr++ - *orgptr++];
}
```

Fig. 13. The body of the hottest loop in the benchmark of h264ref

The interesting thing to note here is that the loop body calls the function “PeYline\_11”. “PeYline\_11” is a function pointer that is set dynamically at run-time, so this function call cannot be inlined by the compiler. Function calls are much faster in 64-bit mode since the calling convention allows up to 8 arguments to be passed through registers (because of the availability of additional registers). In contrast, arguments are passed through the stack in 32-bit mode. As a result, there are more instructions and more memory accesses in the 32-bit version of h264ref, causing about a 2x slowdown for this loop.

Additional registers also help to minimize the number of spills in 64-bit code in this and some other hot spots. The result is a 10% drop in the dynamic instruction count shown

in Figure 5 and a 23% drop in the number of data cache accesses in the 64-bit mode shown in Figure 10.

### E. xalancbmk

xalancbmk is a XSLT processor for rendering XML documents into other types such as HTML. The parsing-centered algorithm used in this benchmark makes intensive usage of linked list data structures and does intensive pointer chasing. Several heavily used classes have almost all (or all) of their data elements stored as pointers. The doubling of the size of the pointer data type in 64-bit mode introduces a performance hit, which is evident in the runtime memory footprint shown in Figure 4 and the data cache request rate shown in Figure 10.

## V. ANALYSIS SUMMARY AND COMPARISON WITH CPU2000

In this section we first summarize the analysis of the five benchmarks as presented in Section IV. We then compare the performance difference of CPU2000 [12] integer benchmarks between 32-bit and 64-bit modes.

Table III shows some key performance metrics (32-bit vs. 64-bit differences) of these five CPU2006 integer benchmarks and a summary of first-order cause that leads to these observed performance differences. Our analysis demonstrates some of the major advantages and pitfalls of the 64-bit computing on x86-64 architecture.

The common traits of these programs that suggest the program can benefit most from x86-64 64-bit mode are: (1) Use of 64-bit integer arithmetic; (2) Presence of loop bodies that require more than a few registers (note that loop unrolling as a common compiler optimization that can also increase register pressure); (3) Many calls to small functions that can be economically inlined.

The major traits of these programs that suggest potential pitfalls for x86-64 64-bit mode are: (1) Memory intensive applications; especially those that have already experienced a high data cache miss rate in a 32-bit environment; (2) Intensive use of long and pointer data types in terms of the amount of memory allocated for such data types and the frequency of their access.

Figure 14 shows a performance comparison of the CPU2000 integer benchmarks as run in 64-bit mode versus in 32-bit mode (CPU2000 is the previous generation of SPEC CPU suite.) The experiment was carried out on the same experimental system as described in Section II. The compiler utilized was also GCC 4.1.1. It is interesting to note that for the CPU2000 integer benchmarks, the performance in 32-bit mode is very close to the performance in 64-bit mode (only lags 0.46% on average.) This is largely due to a single benchmark (mcf) which runs 59% faster in 32-bit mode than in 64-bit mode. In the contrast, CPU2006 revision of mcf obtained “only” a 26% performance advantage when running in 32-bit mode, as shown in Figure 1.

The reason for the smaller performance gap when moving from CPU2000 to CPU2006 is that in the CPU2006 revision of mcf, some fields of the hot data structures `node_t` and

Benchmark	Run time increase	Mem footprint increase	Dynamic inst count decrease	Dcache req rate increase	Cause of performance difference
mcf	26.35%	100.12%	5.74%	33.40%	Larger memory footprint due to use of long and pointer data types in 64-bit mode.
hmmmer	-34.34%	9.84%	8.72%	15.80%	More registers available in 64-bit mode.
libquantum	-35.38%	-31.44%	53.71%	62.50%	Native 64-bit integer arithmetic in 64-bit mode.
h264ref	-35.35%	5.73%	9.96%	22.94%	Faster calling convention (because of more registers) in 64-bit mode.
xalancbmk	13.65%	33.87%	7.60%	28.32%	Larger memory footprint due to pointers in 64-bit mode.

TABLE III

PERFORMANCE DIFFERENCE OF FIVE SPEC CPU2006 INTEGER BENCHMARKS BETWEEN 64-BIT MODE AND 32-BIT MODE (64-BIT MODE RELATIVE TO 32-BIT MODE) IN FOUR METRICS AS WELL AS THE FIRST-ORDER REASONS FOR THESE DIFFERENCES

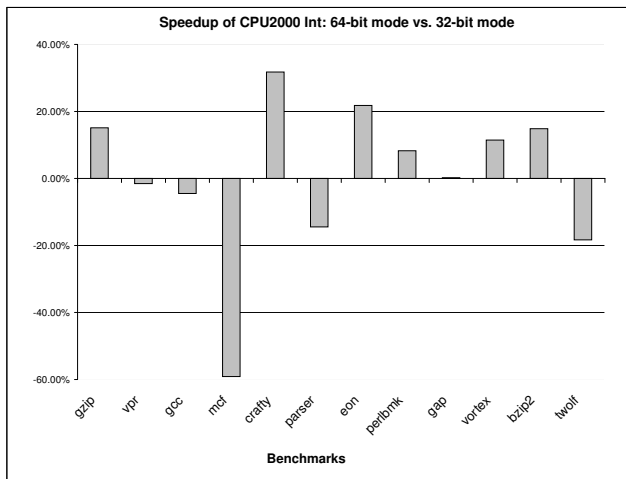


Fig. 14. Speedup of SPEC CPU2000 integer benchmarks running in 64-bit mode vs. in 32-bit mode of x86-64 architecture

arc\_t (as shown in Section II) were changed from ‘long’ to ‘int’. As a result, the memory footprint in 64-bit was reduced, which consequently reduced the performance gap between 64-bit mode and 32-bit mode. This may suggest one way to restructure applications when porting to a 64-bit environment.

## VI. CONCLUSIONS

In this paper, we report and analyze the performance differences observed between the 64-bit and 32-bit modes of the x86-64 architecture. To drive this study, we used the SPEC CPU2006 integer benchmarks. We presented hardware counter data that helps to shed light on the program characteristics that can be exploited by these two modes. We further analyze 5 benchmarks to find the common program traits that benefit one mode over the other. Our analysis data can be utilized by developers and performance engineers to optimize their programs to best exploit the advantages of x86-64 and avoid the potential pitfalls.

In the future, we would like to extend our analysis to the floating point benchmarks of the SPEC CPU2006 suite and some real-world server and desktop applications. Also, we would like to investigate some compiler optimizations that can help mitigate the problems encountered by these benchmarks and applications when running in 64-bit mode.

## ACKNOWLEDGMENT

This work was done when Dong Ye worked in AMD Austin as a co-op engineer. The authors want to thank anonymous reviewers for their suggestions. © 2006 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD logo, and combinations thereof and AMD Athlon 64, AMD-K8 and AMD HyperTransport are trademarks of Advanced Micro Devices, Inc. SUSE® is the registered trademark of Novell, Inc. Linux® is the registered trademark of Linus Torvalds.

## REFERENCES

- [1] AMD. (2006) AMD64 architecture programmer’s manual volume 1. [Online]. Available: <http://developer.amd.com/>
- [2] C. Keltcher, K. McGrath, A. Ahmed, and P. Conway, “The AMD opteron processor for multiprocessor servers,” *IEEE Micro*, vol. 23, no. 2, pp. 66–76, Mar/Apr 2003.
- [3] T. Sweeney. We need 64-bit today. [Online]. Available: <http://slashdot.org/comments.pl?sid=54835&cid=5371889>
- [4] M. K. Base. RAM, virtual memory, pagefile and all that stuff. [Online]. Available: <http://support.microsoft.com/kb/555223>
- [5] SPEC. SPEC CPU2006 benchmark suite. [Online]. Available: <http://www.spec.org/cpu2006/>
- [6] AMD. (2006) Software optimization guide for AMD64 processors. [Online]. Available: <http://developer.amd.com/>
- [7] Novell. SUSE linux. [Online]. Available: <http://www.novell.com/products/suselinux>
- [8] Free Software Foundation. GCC, GNU compiler collection. [Online]. Available: <http://gcc.gnu.org/>
- [9] OProfile. [Online]. Available: <http://oprofile.sf.net/>
- [10] K. McGrath and D. Christie, “The AMD x86-64 ISA: Extending the x86 to 64-bits,” in *Hot Chips 14*, Aug. 2002.
- [11] H264/AVC. [Online]. Available: <http://iphome.hhi.de/suehring/tml>
- [12] SPEC. SPEC CPU2000 benchmark suite. [Online]. Available: <http://www.spec.org/cpu2000/>