Performance Comparison of 32-bit and 64-bit Programs on the AMD64 Architecture

by Wade Spires

December 1, 2004

Table of Contents

List of Tables and Figures

1 Introduction

In recent years, 64-bit architectures have emerged from IBM, Intel, and AMD. The most obvious gain from 64-bit systems over 32-bit systems is the ability to address memory passed the 4 GB limit. This is an important feature for scientific computing, which deals with large data sets, but has become more important to the average user as memory prices fall, larger memory sizes become available, and programs expect more memory. However, this extension is not the only adjustment made from moving to a 64-bit system. In this report, we attempt to determine how performance is affected by moving forward to 64-bit architectures.

2 Test Systems

In the following sections, we provide a summary of the hardware and software used for the experiments. We first list the main hardware and software components. This is followed by the particular benchmark suite used. Finally, the profiling tool employed is described.

2.1 Main Hardware and Software Components

The hardware and software components used are summarized in Tables 1 and 2. Table 1 shows us using the AMD64 architecture for our experiments. AMD64 supports two modes of primary interest—64-bit mode and compatibility mode. With 64-bit mode, AMD64 extends the x86 architecture by adding several improvements:

- 64-bit virtual addresses
- Register extensions:
  - 8 GPRs added (16 total)
  - GPRs widened to 64 bits
  - 128-bit streaming SIMD extensions (SSE) registers added
- 64-bit instruction pointer
- Instruction-relative data-addressing mode

The other notable mode, compatibility mode, provides binary compatibility with existing 32-bit x86 applications. This allows programs to run under 64-bit system software without recompilation. Using both these modes, we can more accurately measure the performance difference between 32-bit and 64-bit versions of the same program by running each version on the same machine.

	Hardware		**Software**
Hardware Vendor:	Advanced Micro Devices	**Operating System:**	Fedora Core 2 (Red Hat) GNU/Linux, kernel 2.6.5
CPU:	AMD Athlon (TM) 64 3200+	**Compiler:**	GNU GCC version 3.3.3
CPU MHz:	2000	**File System:**	ext3
Motherboard:	Gigabyte K8 Triton (GA-K8N Pro)		
Primary Cache:	64 KB (Inst.) + 64 KB (Data) on chip		
Secondary Cache:	1024 KB (I+D) on chip		
Memory:	2x512MB PC3200 DDR SDRAM		
Hard Disk Drive:	Maxtor 6Y080M0, SATA, 7200 rpm		

Table 1 – Components of AMD64 test system

Table 2 shows the other test system we will be using. This system, comprised of a Pentium 4 without Hyper-Threading, is strictly a 32-bit system. Although we would have preferred to use another AMD Athlon processor, one was not available at the time of these tests. Noting that the raw clock speed of the Intel processor (2.4 GHz) is 20% faster than that of the AMD (2.0 GHz), we will be curious to see if AMD's alternate processor labeling (3200+) indeed holds true or if Intel's reliance on clock speed is the more accurate metric.

	Hardware		**Software**
Hardware Vendor:	Intel Corporation	**Operating System:**	Fedora Core 2 (Red Hat) GNU/Linux, kernel 2.6.5
CPU:	Pentium 4	**Compiler:**	GNU GCC version 3.3.3
CPU MHz:	2400	**File System:**	ext3
Motherboard:	Intel D850EMV2		
Primary Cache:	12 k micro-ops (Inst.) + 8 KB (Data) on chip		
Secondary Cache:	512 KB (I+D) on chip		
Memory:	1x512MB PC2100 DDR SDRAM		
Hard Disk Drive:	Western Digital WD800JBRTL, EIDE, 7200 rpm		

Table 2 – Components of Pentium 4 test system

2.2 Benchmark Suite

We use the CPU2000 benchmark developed by the Standard Performance Evaluation Corporation **(**SPEC). We used only a subset of the full suite for our tests—gzip, mcf, vpr, crafty, bzip2, parser, and twolf for the integer benchmarks and ammp, art, equake, and mesa for the floating point benchmarks. We used GCC version 3.3.3 to compile the benchmark programs for both systems. We used the -O3 and -fomit-frame-pointer optimization flags when compiling the programs. The flag -m32 was used to compile programs into 32-bit compatibility mode on the AMD64. This flag was omitted on the natively 32-bit Pentium 4 machine. Table 3 summarizes the benchmarks used and compiler flags used.

Benchmarks (SPEC CPU2000)						
CINT:	gzip	mcf	vpr	bzip2	parser	twolf
CFPU:	ammp	art	equake	mesa		
Compiler Flags (GCC 3.3.3)						
64-bit:	-O3 -fomit-frame-pointer					
32-bit:	-O3 -fomit-frame-pointer -m32					

Table 3 – Selected benchmarks and compiler flags used for tests

2.3 Profiling Tool

We use the publicly available OProfile utility to generate profile data for the benchmarks. It consists of a kernel driver, a daemon for collecting sample data, and several post-profiling tools for interpreting data. OProfile is a statistical continuous profiler. Profiles are generated by regularly sampling the performance counter registers on a CPU and recording the current program counter after a certain number of events have occurred. When running the profiler, two main parameters are specified—an event to monitor and a threshold count. Every time the number of occurrences of an event exceeds the given threshold, an interrupt to software is generated, which can be sampled by OProfile and matched to the corresponding binary image of a process. The events that can be monitored, such as cache accesses and branch mispredictions, are architecture specific as not all systems provide the same hardware performance counter registers. Refer to Appendix B for information on using OPorfile.

Table 1 shows the events we measure for the AMD64 machine. For all the benchmark programs, we are interested in the number of clock cycles, number of instructions, and cache information. For the integer benchmarks, we view the branch information as the next most important data to track. However, for the floating point programs, we measure the number of floating point instructions and dispatch stalls caused by a full floating point unit as we feel this has a strong impact on performance. The counts were selected to be rather high in attempt to reduce the overhead associated with running the profiler. Using higher values sacrifices some accuracy, but too low a value will actually freeze a system due to the large number of interrupts generated.

Event	Count	Description	Benchmark
CPU_CLK_UNHALTED	1000000	Cycles outside of halt state	INT, FPU
RETIRED_INSNS	1000000	Retired instructions	INT, FPU
DATA_CACHE_ACCESSES	100000	Data cache accesses	INT, FPU
DATA_CACHE_MISSES	10000	Data cache misses	INT, FPU
ICACHE_FETCHES	100000	Instruction cache fetches	INT, FPU
ICACHE_MISSES	10000	Instruction cache misses	INT, FPU
RETIRED_BRANCHES	100000	Retired branches	INT
RETIRED_BRANCHES_MISPREDICTED	10000	Retired branches mispredicted	INT
RETIRED_FPU_INSTRS	100000	Retired floating point instructions	FPU
DISPATCH_STALL_FPU	100000	Dispatch stalls when FPU is full	FPU

Table 4 – OProfile events used for AMD64 system

For the Pentium 4, we only measure two events—clock cycles and instructions—as shown in Table 2. One reason is that, unlike with the AMD processor, OProfile uses synthesised events and doesn't provide low-level access to P4 hardware. Also, we feel that the primary events of interest are the number of clock cycles and instruction count.

Event	Count	Description	Benchmark
GLOBAL_POWER_EVENTS	1000000	Cycles outside of halt state	INT, FPU
INSTR_RETIRED	1000000	Retired instructions	INT, FPU

Table 5 – OProfile events used for Pentium 4 system

3 Benchmark Experiments

We provide several results from running the benchmarks. First, the SPEC base times are shown. Using these times as a base, further analysis is conducted using profiled data. Note that all results shown are the average of three runs.

3.1 SPEC Experiments

Figure 2 shows the SPEC base numbers we recorded from the integer benchmarks for each system. As shown, the results do not show a consistent performance increase from compiling in 64-bit mode as opposed to 32-bit mode. While the benchmarks gzip, vpr, crafty, and bzip2 yield improved performance in 64-bit mode, the other benchmarks—mcf, parser, and twolf—show degraded performance when run in 64-bit mode instead of 32-bit mode. We see that the benchmark crafty, which relies primarily on 64-bit integer operations, has the largest relative performance increase in 64-bit mode. This leads us to suspect that some programs might see a larger increase in speed if they were designed or programmed to take advantage of the new 64-bit features. Perhaps, then, the relatively poor performance for the mcf program in 64-bit mode could be improved somewhat. We also note that despite the higher clock rate, the 2.4 GHz Pentium 4 is in all cases significantly slower than the 2 GHz AMD processor. We guess that some other architectural features of the AMD64, such as the much larger cache or shorter pipeline, compensate for the lower clock rate.

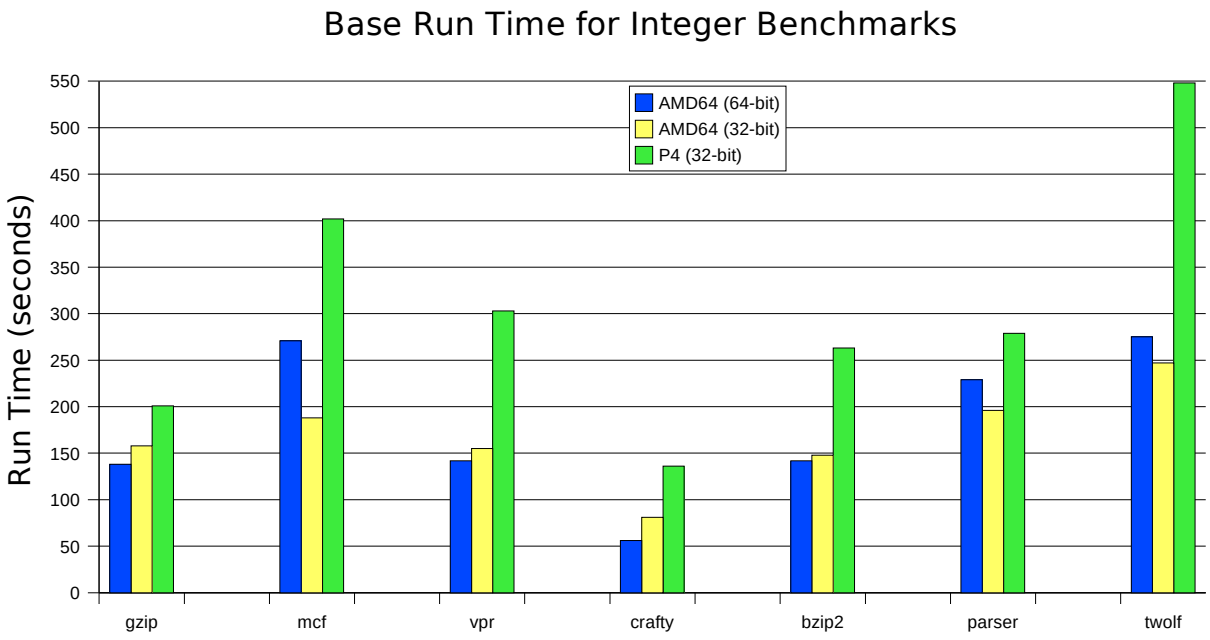

Figure 1 – Base run time for integer benchmarks given by SPEC tools

Figure 2 shows the base numbers for the floating point benchmarks. We see gains in performance for all the floating point benchmarks except for equake where the results are nearly equal. The art benchmark especially sees a very large performance boost with an 80% decrease in run time for the 64-bit version. We believe the 64-bit nature of most floating point data allowed for more efficient operation in 64-bit mode.

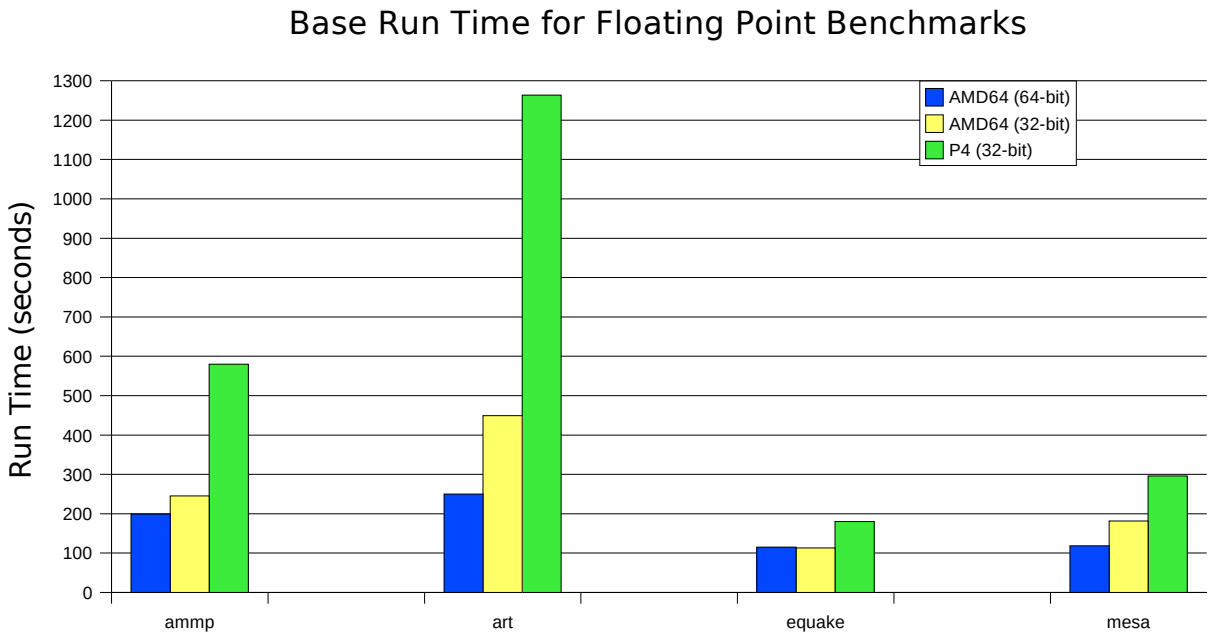

Figure 2 – Base run time for floating point benchmarks given by SPEC tools

Table 6 shows the speedup gained from going from a 32-bit compiled application to a 64-bit compiled application on the AMD64 system using the SPEC base run times from Figures 1 and 2 where

Speedup = 32-bit Execution Time / 64-bit Execution Time.

While the speedup for the integer benchmarks is negligible, the floating point gains are significant.

	gzip	mcf	vpr	crafty	bzip2	parser	twolf	Average
Speedup	1.14	0.69	1.09	1.45	1.04	0.86	0.89	1.02
	ammp	**art**	**equake**	**mesa**				**Average**
Speedup	1.23	1.8	0.98	1.53				1.39

Table 6 – Speedup from using 64-bit mode instead of 32-bit mode

3.2 Profiling Experiments

Having performed some preliminary analysis of performance using the SPEC tools, we now use the profiler to gain deeper insight into performance issues. First, we look at the number of clock cycles and instructions used by each benchmark. Then we analyze the data and instruction cache performance. Finally, we assess the role of branches on performance for the integer benchmarks and dispatch stall cycles for the floating point benchmarks. Our data is also summarized in tabular form in Appendix A.

3.2.1 Clock Cycles and Instruction Count

The number of clock cycle samples is shown in Figure 3 for the integer benchmarks. Note that only the number of samples are given, so the actual number of clock cycles for gzip, for instance, is 263,136 * 1,000,000, not just 263,136 in 64-bit mode. We see that the profiling results reflect the SPEC results in that gzip, vpr, crafty, and bzip2 show improved results in 64-bit mode, while mcf, parser, and twolf have worse results in 64-bit mode.

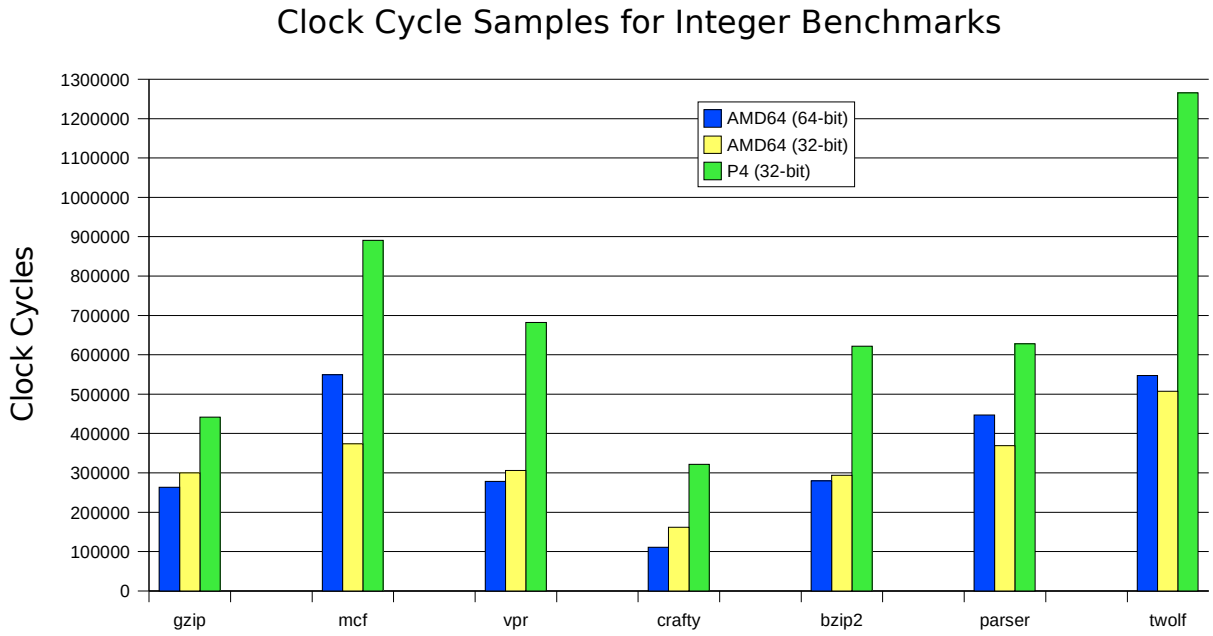

Figure 3 – Number of clock cycles used by each integer benchmark

Using these numbers, we can calculate the CPU time using the following formula:

CPU Time = Number of Clock Cycles / Clock Rate.

For clock rate, we multiply the clock rate for each processor in MHz (2000 and 2400) by 10^6 to express CPU time in seconds. The number of clock cycles, obviously, comes from Figure 3, but we must multiply by 1,000,000 since that is our sampling count. As 1,000,000 equals 10^6, we simply take our sample count and divide by the clock rate to obtain the CPU time in seconds. This gives us another showing of run time in Figure 4. We see that the numbers are very similar to the base run time generated by SPEC in Figure 1. The times here are possibly larger due to the overhead required by the profiler. We do, however, see the same trends holding for the benchmarks seen in Figure 1.

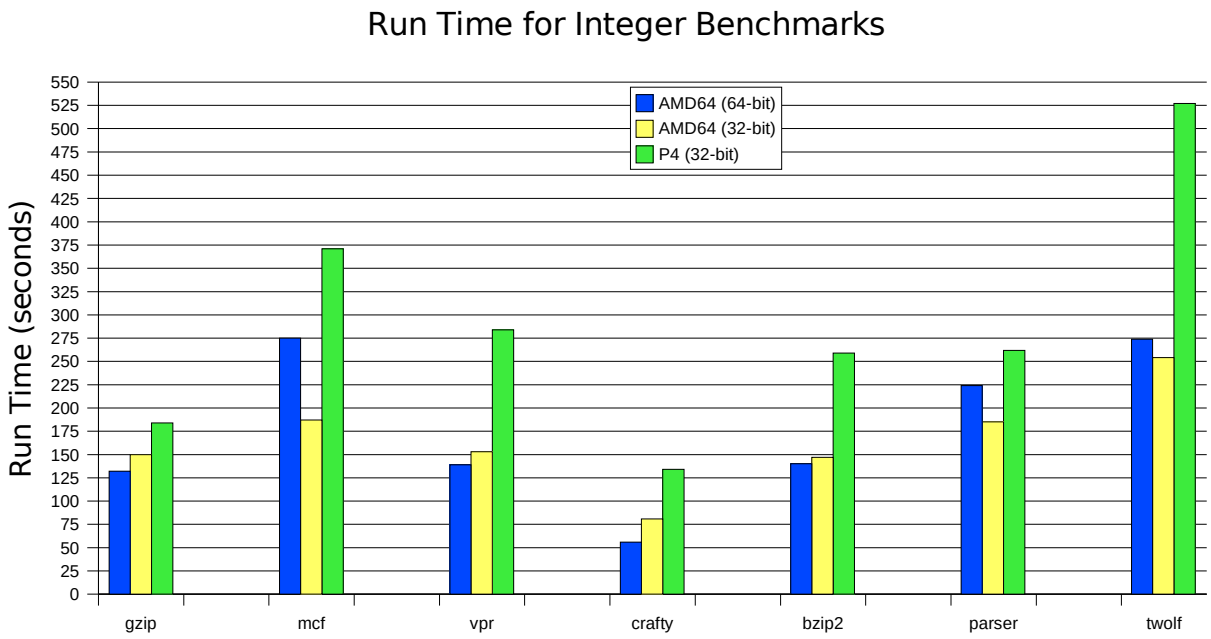

Figure 4 – Run time for integer benchmarks based on profile data

Figure 5 shows the clock cycles for the floating point benchmarks, while Figure 6 shows the run time using this data. We compute CPU time for the floating point benchmarks in the same way as for the integer programs. Again, we note the similarity between these times and the ones produced by SPEC.

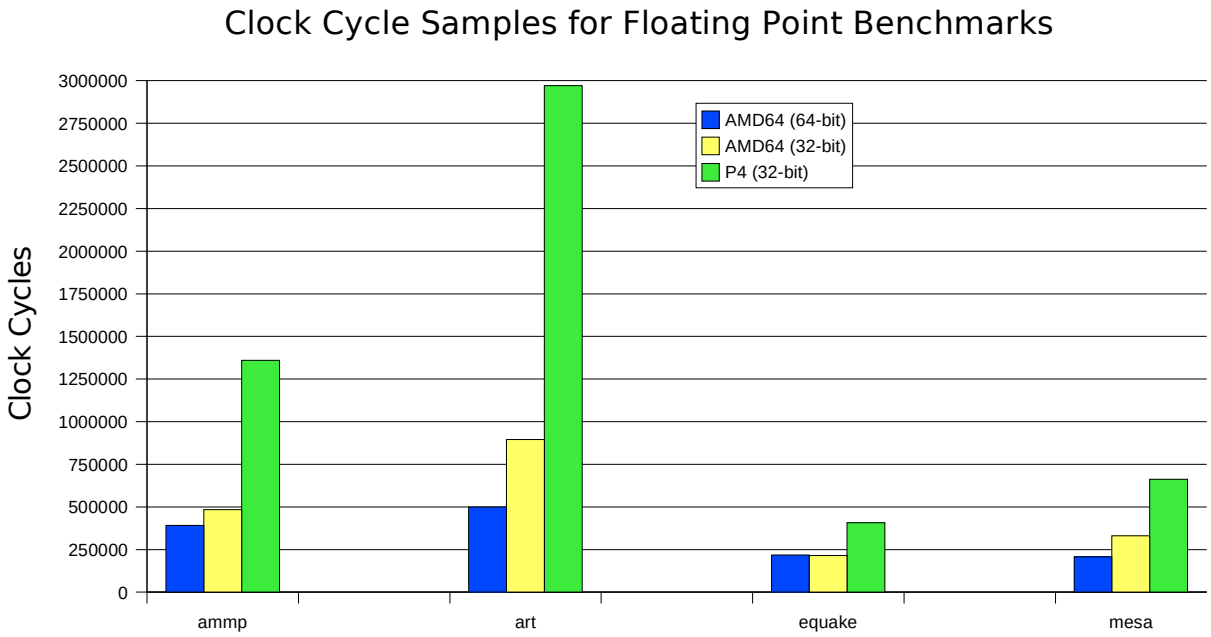

Figure 5 – Number of clock cycles used by each floating point benchmark

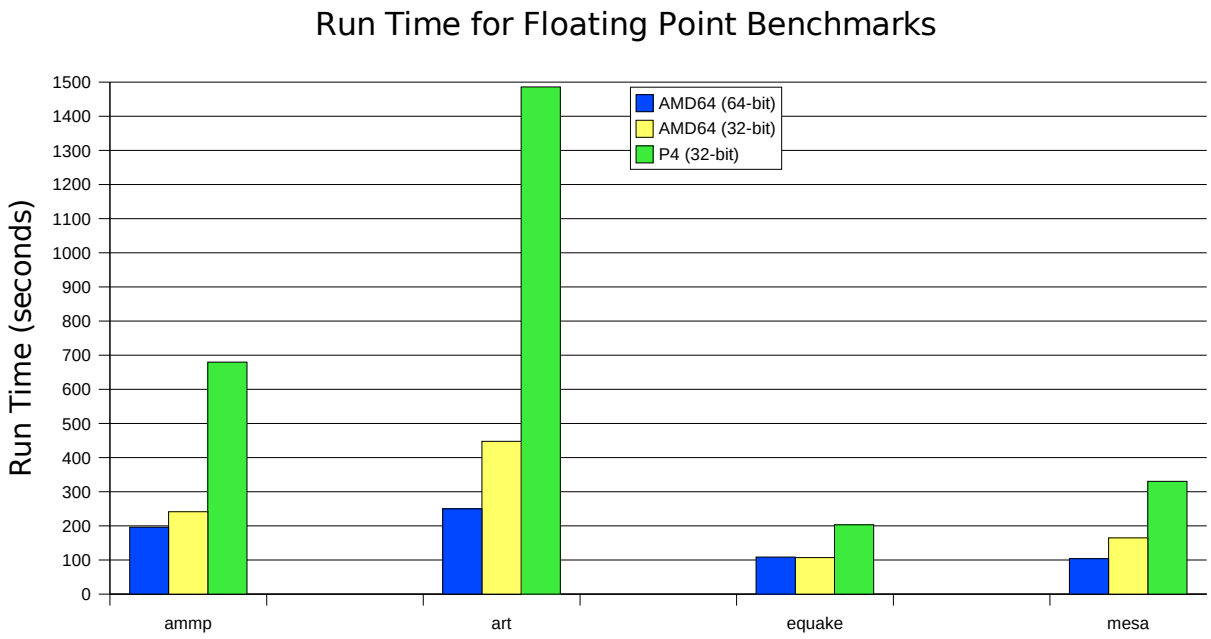

Figure 6 – Run time for floating point benchmarks based on profile data

Using the profile data, we summarize the speedup from using 64-bit mode instead of 32-bit mode for the programs in Table 7. The speedup seen here is only slightly higher than that seen in Table 6.

	gzip	**mcf**	**vpr**	**crafty**	**bzip2**	**parser**	**twolf**	**Average**
Speedup	1.14	0.68	1.1	1.45	1.05	0.83	0.93	1.03
	ammp	**art**	**equake**	**mesa**				**Average**
Speedup	1.23	1.79	0.98	1.59				1.4

Table 7 – Speedup from using 64-bit mode instead of 32-bit mode based on profile data

Figures 7 and 8 show the dynamic instruction counts for the integer and floating point benchmarks, respectively. By retired instruction, we mean instructions that were both fetched and executed, e.g., instructions that were fetched but discarded due to a branch misprediction are not counted. Additionally, the number of instructions refers to actual x86 instructions, not the RISC-like micro-ops that each instruction is translated to.

We see that for a given integer benchmark that the number of instructions are nearly the same across the different systems and modes. The integer benchmark crafty and all the floating point benchmarks show fewer instructions being executed in 64-bit mode. We theorize that, since these programs deal with 64-bit data types, in 32-bit mode additional data fetches must occur in order to retrieve one piece of data. This implies an additional reference to memory can be removed for 64-bit mode.

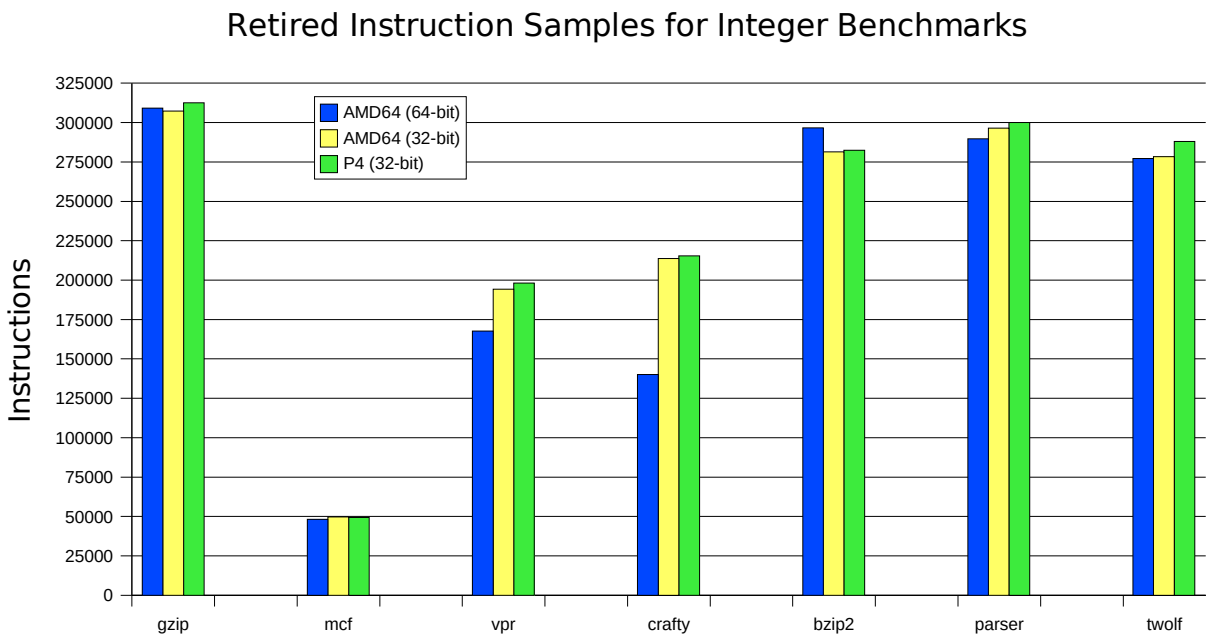

Figure 7 – Number of instructions executed for integer benchmarks based on profile data

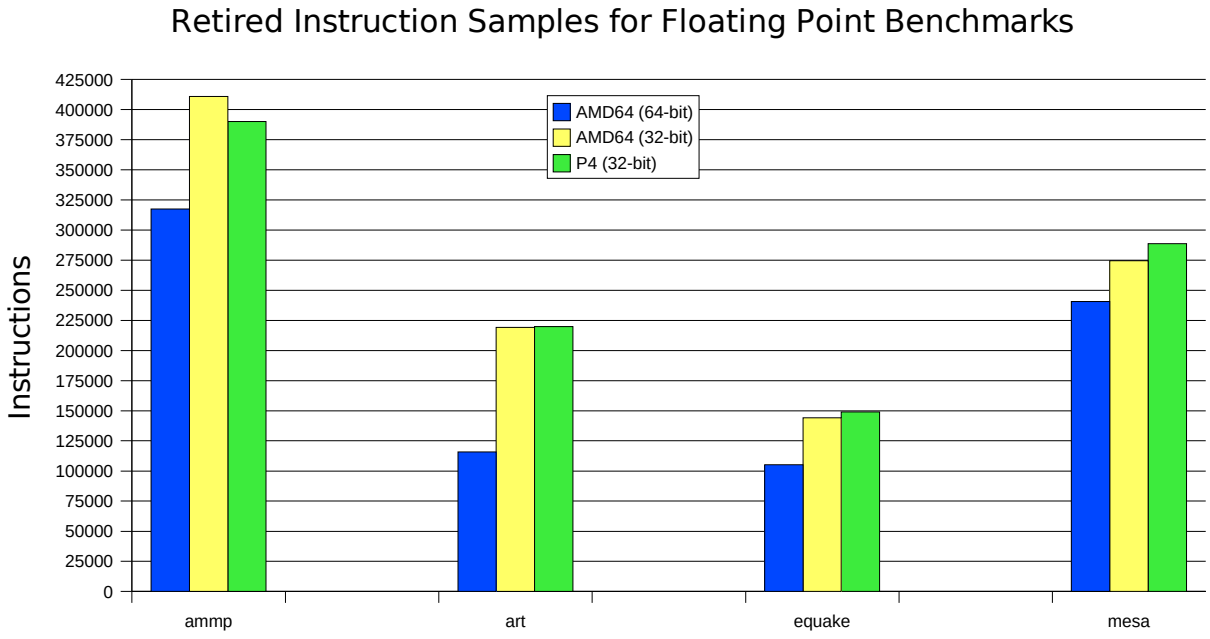

Figure 8 – Number of instructions executed for floating point benchmarks based on profile data

3.2.2 Cache Performance

Figure 9 shows the total number of data cache accesses. For nearly every benchmark, the number of accesses is reduced for the 64-bit programs. This is most pronounced in the crafty and floating point benchmarks, supporting our theory that the 64-bit hardware reduces the number of data fetches for those programs that deal primarily with 64-bit data. For the other benchmarks, we guess that the 8 additional GPRs allow for fewer data fetches also.

Table 8 shows the data cache miss rate for each benchmark. Although the results show the 64-bit compiled benchmarks having higher miss rates, this should only be considered in light of the total number of accesses. For instance, the art benchmark has nearly twice the miss rate in 64-bit mode, but the total number of misses by 32-bit art is about twice that of the 64-bit version since the 32-bit version performs so many more accesses. Another observation we make is that mcf has a significantly higher miss rate in 64-bit mode (32% versus 23%). Since the total number of data cache accesses are approximately the same, we see one cause of the reduced performance for this program in 64-bit mode. In fact, referring to Figure 10 listing total number of misses, we see more cache misses for those programs having longer execution times in 64-bit mode. As a benchmark such as mcf is dominated by loads, we again see this as a strong indicator to where the bottleneck lies. As for why 64-bit mode causes this problem, we guess that data for mcf has lower spatial locality and that 64-bit mode pulls larger sized blocks into the cache, which increases miss rate.

Another interesting case is equake, which had a larger number of misses in 64-bit mode but with only a marginal degrade in performance. Although equake has a high percentage of load instructions, we believe the cycles wasted resolving cache misses is balanced out by the longer latency floating point operations, such as multiplication.

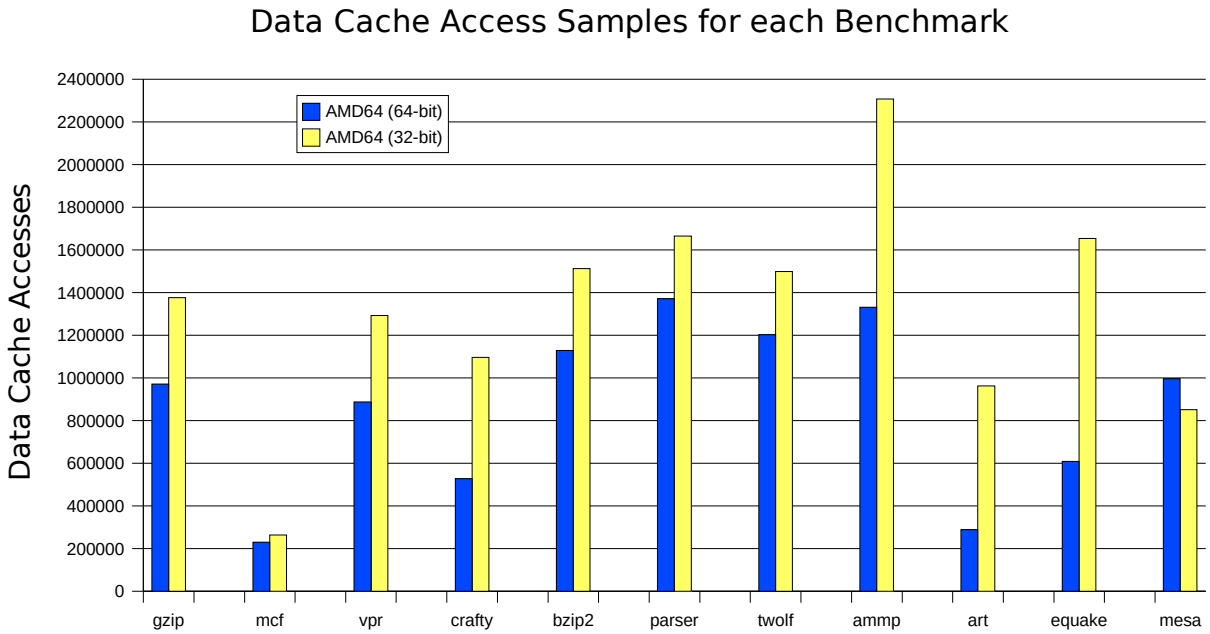

Figure 9 – Number of data cache accesses

	gzip	mcf	vpr	crafty	bzip2	parser	twolf	ammp	art	equake	mesa
AMD64 (64-bit)	4.09	31.83	4.16	0.88	2.04	3.9	6.21	4.05	38.95	5.58	0.03
AMD64 (32-bit)	2.86	22.52	2.64	3.12	1.53	1.72	4.51	2.32	21.4	1.82	0.34

Table 8 – Data cache miss rate percentages

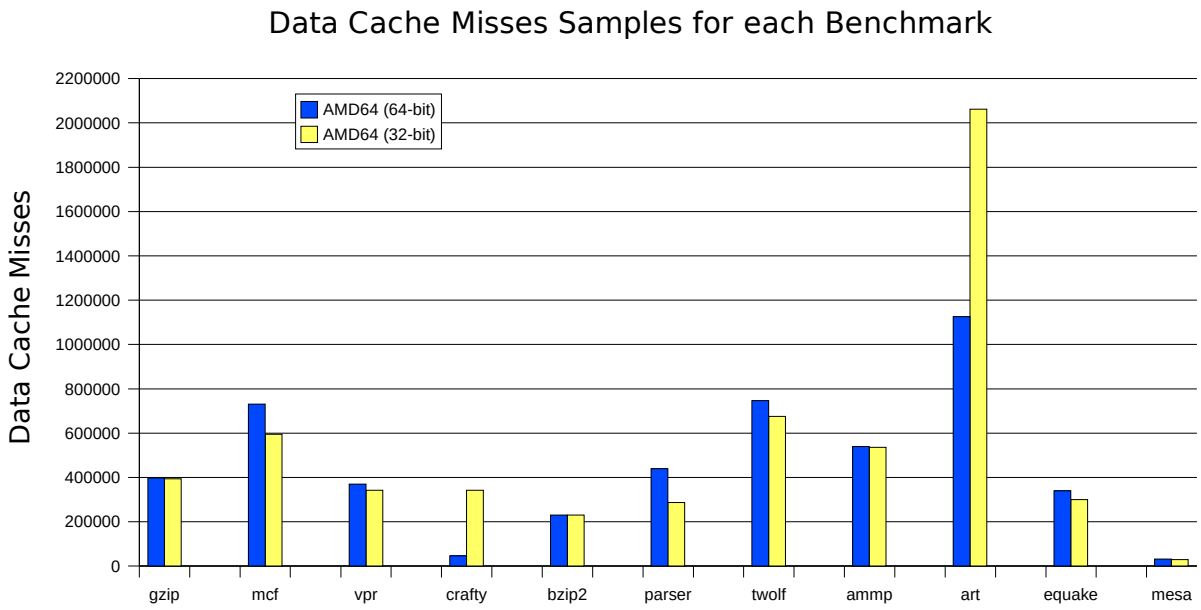

Figure 10 – Number of data cache misses

We now take a look at the number of fetches from the instruction cache. Figure 10 shows that the number of fetches for both modes are rather similar. The largest difference is with the program art, which also had significantly fewer instructions executed as shown above in Figure 8.

We do not provide the number of instruction cache misses nor miss rates as all were insignificant compared to the number of fetches—almost 1% for one benchmark and usually much less. One observation, however, was that crafty has a large difference in the number of misses between modes with 32-bit mode having about 36 times the misses as 64-bit mode. Mesa, also, has a huge number of misses in 32-bit mode relative to 64-bit mode with over 260 times as many fetch misses. Despite these large differences in cache misses, we do not see as large a difference in execution time since these misses only account for less than 1% of all instruction cache fetches.

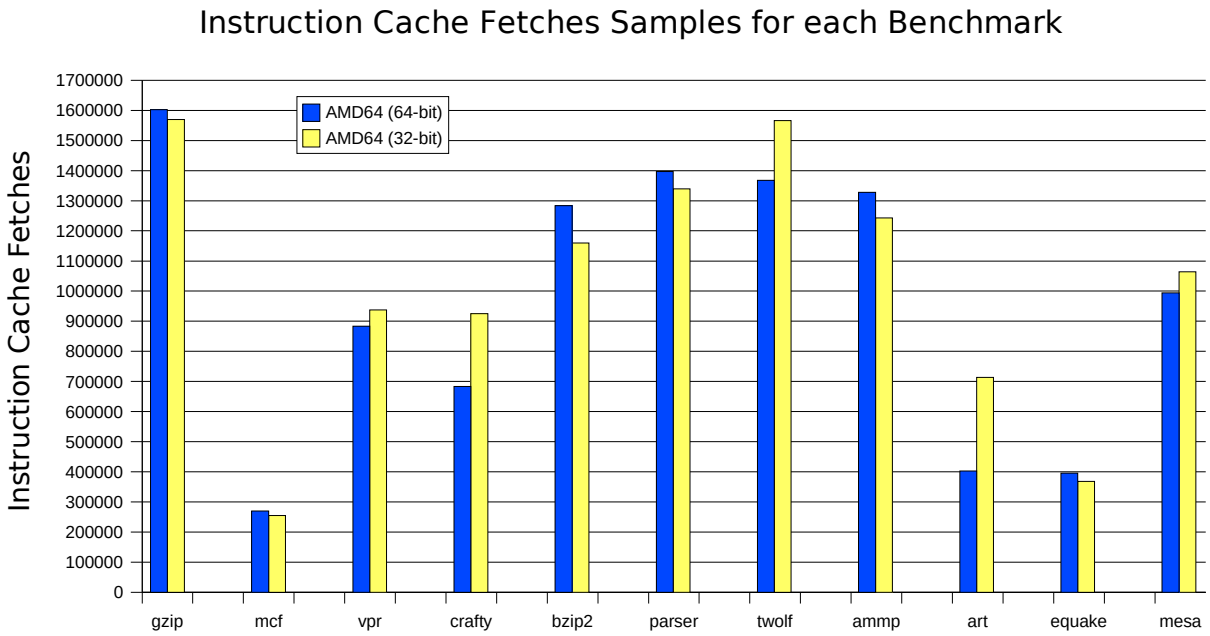

Figure 11 – Number of instruction cache fetches

3.2.3 Branches and Floating Point Instructions

For the integer benchmarks, we look at branch prediction as that is a key component of efficient architectures. However, we do not expect branch performance to be much affected by the bit mode of the program. We observe in Figure 11 both the 32-bit and 64-bit versions of each program having a similar number of branches, which is unsurprising. We do unexpectedly observe somewhat fewer branch mispredictions for the 64-bit programs. Although this must help performance for the 64-bit applications, we cannot readily determine why this would occur. It is possible that 64-bit mode looks at more bits in the instruction pointer so that a branch counter corresponds to only one branch instruction, giving greater accuracy. This is just speculation, however. We provide misprediction rates in Table 9.

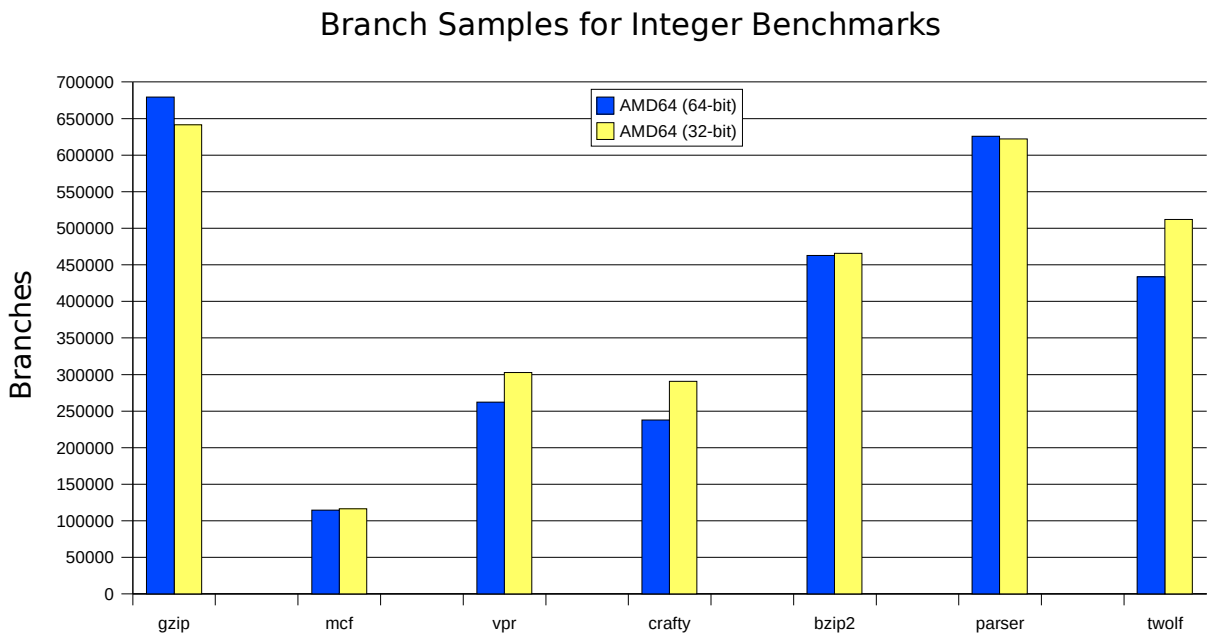

Figure 12 – Number of branches executed

	gzip	mcf	vpr	crafty	bzip2	parser	twolf
AMD64 (64-bit)	3.71	4.86	8.64	6.22	5.45	4.71	10.06
AMD64 (32-bit)	5.24	5.07	8.58	6.5	5.47	5.01	11.53

Table 9 – Branch misprediction rate percentages

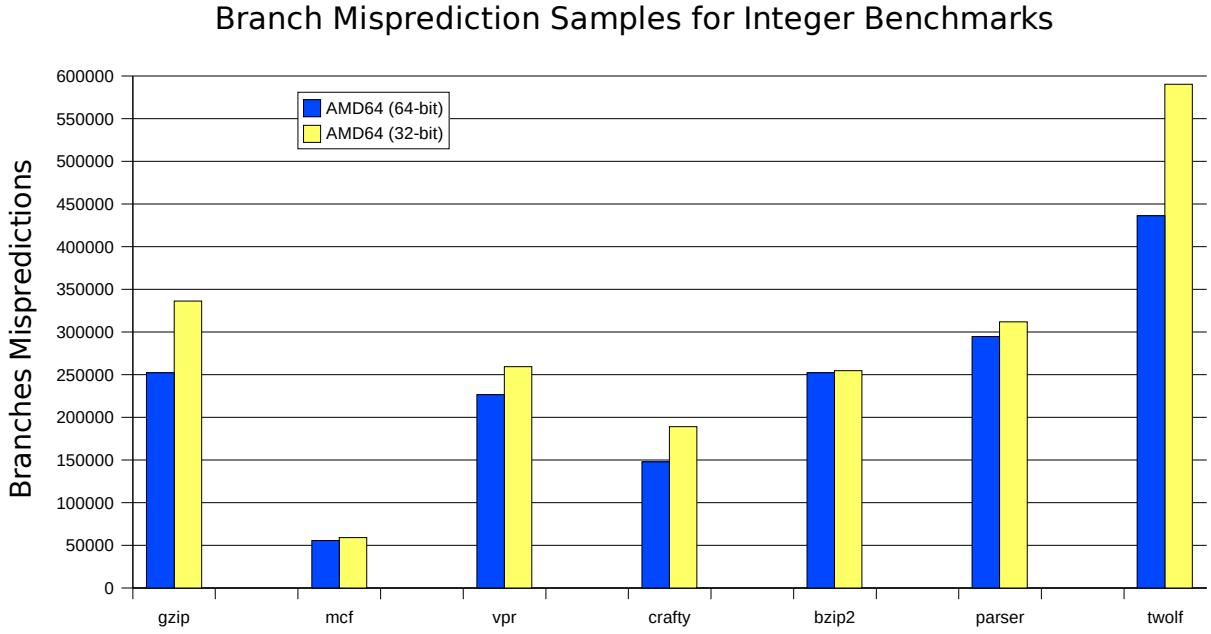

Figure 13 – Number of mispredicted branches

For the floating point benchmarks, we see from Figure 12 that in all cases the number of floating point instructions executed is reduced in 64-bit mode. As with the reduction in total instructions seen in Figure 8, we believe this is due to fewer memory fetches required to retrieve a 64-bit wide operand. We also see an increase in the number of dispatch stalls due to a full floating point unit for the 32-bit programs. Possibly the dispatch unit stalls from data being by-passed to an execution unit waiting for its operands to be loaded from memory.

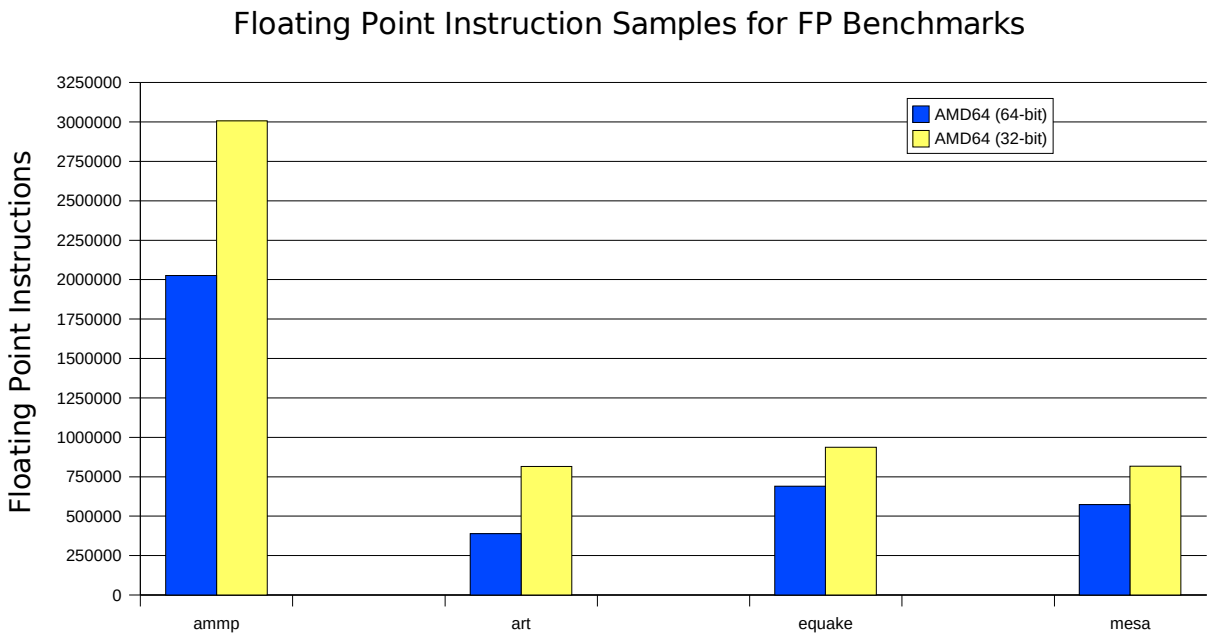

Figure 14 – Number of floating point instructions executed

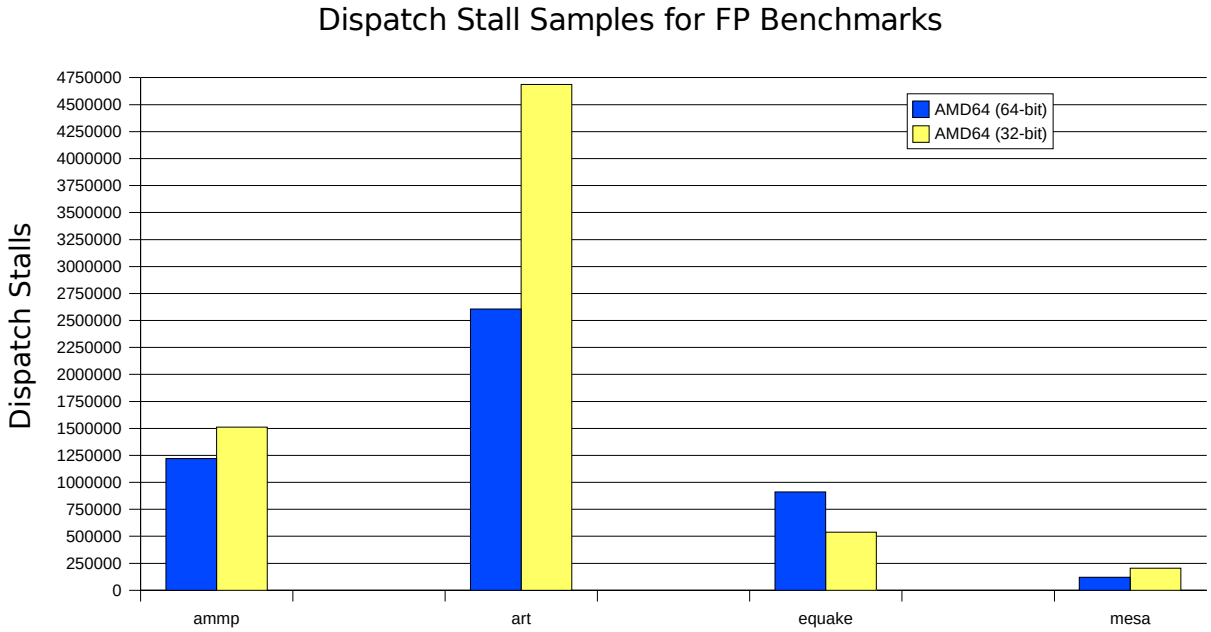

Figure 15 – Number of dispatch stalls caused by full FP units

4 Conclusions

Having completed our experiments, several conclusions may be drawn. The main finding is that 64-bit computing does not automatically translate to a marked improvement in performance. If a program does not make use of 64-bit data types, then we will probably see little benefit in execution time. In fact, the execution speed may be worse, most likely due to issues with cache utilization. However, some programs that primarily operate on 32-bit data may still see a moderate speedup under the AMD64 architecture, most likely due to the additional GPRs added to the x86 instruction set. For programs utilizing 64-bit data types, such as many floating point heavy scientific applications, then an approximate 40% boost in performance can be expected, which provides a strong incentive to invest in a 64-bit system.

4.1 Future Work

Future experiments regarding 32-bit versus 64-bit might encompass a few improvements. First, while we were able to see noticeable patterns in the programs under study, we would like to utilize the full SPEC benchmark suite for our tests. Next, a larger selection of hardware would be useful. We would like to compare these results to the other main 64-bit architectures, such as IBM's PowerPC and Intel's Itanium processors. For comparison against strictly 32-bit machines, we would rather use another AMD processor, such as the Athlon 3000+, as the cache design, branch prediction, and performance counters would be more similar to the 3200+ we tested here. Finally, we would like to see the compiler's role in this issue by testing several different compilers, such as Microsoft Visual Studio or Intel's compiler.

5 References

[1] "64-bit." Internet: http://en.wikipedia.org/wiki/64-bit, Nov. 27, 2004 [Nov. 13, 2004].

[2] Advanced Micro Devices, Inc. "AMD64 Architecture Programmer's Manual Volume 2: System Programming." Sept. 2003.

[3] Andrew Josey. "Data Size Neutrality and 64-bit Support." Internet: http://www.usenix.org/publications/login/standards/10.data.html, Dec. 4, 1997 [Nov. 25, 2004].

[4] Dileep Bhandarkar and Douglas W. Clark. "Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization." *ACM SIGARCH Computer Architecture News.* [Online] 19(2). Available: http://portal.acm.org/citation.cfm?id=106975.107003 [Oct. 15, 2004].

[5] John Hennessy and David Patterson. *Computer Architecture: A Quantative Approach.* Palo Alto: Morgan Kaufmann, 2002.

[6] "OProfile – A System Profiler for Linux (News)." Internet: http://oprofile.sourceforge.net/news, Sep. 13, 2004 [Oct. 25, 2004].

[7] "SPEC – Standard Performance Evaluation Corporation." Internet: http://www.spec.org, Nov. 23, 2004 [Nov. 1, 2004].

Appendix A – Summary of Data Collected

We present a summary of the numerical data collected from OProfile in this section.

Threshold Count:	1000000	1000000	100000	10000	100000	10000	100000	10000
Event:	CPU_CLK_UNHALTED	RETIRED_INSNS	DATA_CACHE_ACCESSES	DATA_CACHE_MISSES	ICACHE_FETCHES	ICACHE_MISSES	RETIRED_BRANCHES	RETIRED_BRANCHES_MISPREDICTED
gzip:	263136	309184	971881	397984	1602179	50	679291	252150
mcf:	549497	48294	229722	731153	269706	38	114652	55674
vpr:	278453	167753	888449	369611	883175	55	262192	226624
crafty:	111020	140121	527893	46644	682839	914	237744	147952
bzip2:	279400	296598	1128709	230667	1283855	35	462684	252187
parser:	447013	289800	1372123	439856	1397294	690	625680	294759
twolf:	547327	277140	1202952	747320	1367962	199	433704	436384

Threshold Count:	1000000	1000000	100000	10000	100000	10000	100000	100000
Event:	CPU_CLK_UNHALTED	RETIRED_INSNS	DATA_CACHE_ACCESSES	DATA_CACHE_MISSES	ICACHE_FETCHES	ICACHE_MISSES	RETIRED_FPU_INSTRS	DISPATCH_STALL_FPU
ammp:	392329	317436	1332155	539837	1327628	125	2026517	1219803
art:	499408	115840	289104	1126093	402582	59	389659	2605307
equake:	218870	105293	609184	340095	395197	22	690901	910738
mesa:	207554	240685	996238	31443	994534	518	573336	121267

Table 10 – Samples gathered from OProfile for AMD64 in 64-bit mode

Threshold Count:	1000000	1000000	100000	10000	100000	10000	100000	10000
Event:	CPU_CLK_UNHALTED	RETIRED_INSNS	DATA_CACHE_ACCESSES	DATA_CACHE_MISSES	ICACHE_FETCHES	ICACHE_MISSES	RETIRED_BRANCHES	RETIRED_BRANCHES_MISPREDICTED
gzip:	299953	307387	1376253	394103	1569501	64	641607	336162
mcf:	373575	49707	264375	595383	254608	19	116648	59146
vpr:	305731	194217	1292560	341667	937655	45	302555	259467
crafty:	161789	213800	1096639	49142	925110	33219	290755	188955
bzip2:	293944	281515	1512500	230742	1159546	39	465894	254740
parser:	369014	296480	1665304	286982	1339458	746	622154	312010
twolf:	507275	278443	1498662	676264	1566129	356	512042	590284

Threshold Count:	1000000	1000000	100000	10000	100000	10000	100000	100000
Event:	CPU_CLK_UNHALTED	RETIRED_INSNS	DATA_CACHE_ACCESSES	DATA_CACHE_MISSES	ICACHE_FETCHES	ICACHE_MISSES	RETIRED_FPU_INSTRS	DISPATCH_STALL_FPU
ammp:	483994	410886	2308546	536192	1243256	121	3006379	1510555
art:	895548	219159	963490	2061734	713675	88	815022	4687688
equake:	214915	144248	1654664	29220	368151	16	936688	537223
mesa:	330372	274517	851578	300452	1064572	92397	817249	203634

Table 11 – Samples gathered from OProfile for AMD64 in 32-bit mode

Threshold Count:	1000000	1000000
Event:	GLOBAL_POWER_EVENTS	INSTR_RETIRED
gzip:	441634	312568
mcf:	890681	49433
vpr:	682295	198219
crafty:	321363	215415
bzip2:	621767	282513
parser:	628023	300067
twolf:	1265446	288098
ammp:	1359233	390054
art:	2971109	219834
equake:	662880	148961
mesa:	406642	288732

Table 12 – Samples gathered from OProfile for Pentium 4

Appendix B – Using OProfile

In this section, we describe how to use OProfile to collect program data. First, we discuss how to install OProfile in case it is not available on a GNU/Linux system. Then we give the most common commands for running OProfile. Both input and output are in `monospace` font with input also in **`bold`**. The character '>' is used as the command prompt in the examples.

B.1 Installing OProfile

While OProfile may already be installed on your system, as was the case with the AMD64 machine, the program may not be available initially, such as with the Pentium 4. Even if the Linux distribution includes OProfile, you may still have to follow these steps as it still may not be a loadable module or compiled into the kernel. Unfortunately, as OProfile works at a rather low system level, direct kernel support must given, which implies that the kernel source may need to be recompiled to use OProfile. We will try to achieve this as simply as possible.

Switch to the root user and check if OProfile is already installed:

```
> su
> opcontrol -l
```

If you see the following message, then you will need to continue on and compile the kernel:

```
FATAL: Module oprofile not found.
FATAL: Module oprofile not found.
Kernel doesn't support oprofile
```

Otherwise, if a list of several events to monitor are listed, then OProfile is already configured to run on your system, and you can skip this section.

As root, change to the directory containing the kernel source code:

```
> su
> cd /usr/src/linux-x.x.x
```

Configure the kernel options:

```
> make menuconfig
```

A text-base menu will appear and you will need to select a few options. Find and select yes to the following options:

```
Processor type and features -> Local APIC support for uniprocessors
Processor type and features -> Local IO-APIC support for uniprocessors
Profiling support -> Profiling support
Profiling support -> OProfile system profiling
```

Exit and confirm that you would like to save your new kernel configuration.

Compile the kernel code and modules, install the modules, and install the final executable into the boot partition (this may take a long time as hundreds of source files will be compiled):

```
> make bzImage modules modules_install install
```

Once this is done, a new bootable sector, probably called vmlinuz-x.x.xcustom is created in /boot. The old vmlinuz is not removed, so even if something went wrong with the build, you will still be able to boot your old kernel. Finally, reboot the system and, assuming the use of a bootloader such as GRUB,

a new option for a custom linux kernel should appear along with the old kernel choice. Select to boot the custom kernel. You should now be able to use the OProfile tool set.

B.2 Basic Usage

We give an example of using OProfile through a command console with the assumption that OProfile is already available and working on your system (see B.1 if this is not the case).

Gain root access to run the main tool, opcontrol:

```
> su
Password:
```

Print basic usage:

```
> opcontrol
opcontrol: usage:
   -l/--list-events list event types and unit masks
   -?/--help        this message
   -v/--version     show version
...
```

Monitor linux kernel (only setup one time):

```
> opcontrol --vmlinux=/boot/2.6.0/vmlinux
```

Or do not monitor linux kernel (this must be set if only vmlinu**z** is available):

```
> opcontrol —-no-vmlinux
```

List all events that can be monitored:

```
> opcontrol -l
oprofile: available events for CPU type "AMD64 processors"

CPU_CLK_UNHALTED: (counter: all)
      Cycles outside of halt state (min count: 3000)
RETIRED_INSNS: (counter: all)
        Retired instructions (includes exceptions, interrupts, re-syncs)
...
```

Monitor an event using a threshold count of 100000:

```
> opcontrol —-event=CPU_CLK_UNHALTED:100000
```

Start profiling:

```
> opcontrol —-start
Using 2.6+ OProfile kernel interface.
Using log file /var/lib/oprofile/oprofiled.log
Daemon started.
Profiler running.
```

View profile results (any user can run this command—not just root):

```
> opreport
CPU: AMD64 processors, speed 2009.15 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Cycles outside of halt state) with a unit mask of
```

```
0x00 (No unit mask) count 100000
CPU_CLK_UNHALT...|
  samples|      %|
------------------
    28665 44.3312 no-vmlinux
     4690  7.2532 libsal.so.3.1.0
     2821  4.3628 libgcc_s-3.3.3-20040413.so.1
     2076  3.2106 Xorg
   ...
```

Note that you will get this error message if OProfile hasn't had enough time to generate data:

```
opreport error: No sample file found: try running opcontrol --dump
or specify a session containing sample files
```

Stop profiling:

```
> opcontrol --stop
Stopping profiling.
```

Reset count to zero for number of samples collected:

```
> opcontrol --reset
Signalling daemon... done
```

Close background daemon (must also be done before setting new events to monitor):

```
> opcontrol --shutdown
Stopping profiling.
Killing daemon.
```