Sim-alpha: a Validated, Execution-Driven Alpha 21264 Simulator

Rajagopalan Desikan[⋆] Doug Burger[†] Stephen W. Keckler[†] Todd Austin[‡]
[†]Department of Computer Sciences [⋆]Department of Electrical and Computer Engineering
The University of Texas at Austin
[‡] Department of Electrical Engineering and Computer Sciences
The University of Michigan at Ann Arbor

Department of Computer Sciences
Tech Report TR-01-23
The University of Texas at Austin

ABSTRACT

This technical report describes installation, use, and design of *sim-alpha*, an execution driven Alpha 21264 simulator. To increase simulator accuracy, we have incorporated many of the low level features found in the Alpha 21264. When compared to a hardware 21264 implementation, *sim-alpha* achieves 2% error across a suite of microbenchmarks designed to stress the various microarchitectural features in the simulator. The error across the 10 SPECINT 2000 benchmarks is 6.6% and the 12 SPECFP 2000 benchmarks is 21%, with the net error being 15% across the 22 of the 26 SPECCPU 2000 benchmarks.

1 Introduction

The computer architecture community relies heavily on simulators to evaluate new ideas. Publicly available simulators like SimpleScalar [1], Rsim [10], Trimaran [5], and SimOS [11] are widely used and shared by researchers, and numerous papers have been published using the results from these tools. However, few of these tools have been compared against actual hardware. In this report, we describe *sim-alpha*, a validated, execution driven, Alpha 21264 processor simulator. *sim-alpha* was written by extending the SimpleScalar [1] tool suite.

sim-alpha models both the implementation constraints, as well as the performance-improving low level features in the 21264. The simulator includes flags which allows the user to enable and disable these features to study their influence. The simulator allows the user to vary the different parameters of the processor such as the issue queue sizes, the fetch width, and the reorder buffer size. *sim-alpha* achieves 2% error across the set microbenchmarks we used for the validation, and 15% across a set of 22 macrobenchmarks from the SPECCPU 2000 suite. The error across the 10 SPECINT 2000 benchmarks is 6.6%.

The rest of the report is organized as follows. Section 2 describes how to obtain and build *sim-alpha*. Section 3 describes our target system that includes the Alpha 21264 processor, the DS-10L Alphaserver system against which we validate *sim-alpha*, the Digital Continuous Profiling Infrastructure tool set for measuring performance of programs on the native DS-10L system, and the microbenchmarks we used for validating the microarchitecture and memory system in *sim-alpha*. We present *sim-alpha* error across our suite of microbenchmarks and macrobenchmarks in Section 4, and describe usage and internal workings of the tool in Section 5. Finally, Section 6 summarizes our work and suggests future enhancements.

2 Obtaining *sim-alpha*

The *sim-alpha* simulator source code is available as a tar gzipped file through the world wide web at :

`http://www.cs.utexas.edu/users/cart/code/alphasim-1.0.tgz`

The microbenchmarks used for the validation can be obtained from :

`http://www.cs.utexas.edu/users/cart/code/microbench.tgz`

The SPECCPU 2000 benchmark binaries can be obtained from :

`ftp://ftp.simplescalar.org/pub/benchmarks/spec2000/spec2000alpha.tar.gz`

sim-alpha currently runs only on x86/Linux boxes; Since it does not currently have cross-endian support, it cannot run on big-endian machines. The system call support on *sim-alpha* also currently supports only Linux calls. To build the simulator, uncompress the *tgz* file and type *make* in the resulting `alphasim` directory to build *sim-alpha*

```
tar -xvzf alphasim-1.0.tgz
cd alphasim-1.0
make
```

The `alphasim/tests` directory contains compiled test binaries. The simulator uses the SimpleScalar 3.0 Alpha front end emulator, so it can run any binary compiled for the Alpha ISA. *sim-alpha* takes command line arguments and also accepts arguments in a file. The simulator can be compiled in three modes:

1. *Normal mode* where it includes all Alpha 21264 features. This is the default mode. Type
   `make <sim-alpha>`

2. *Flexible mode* where the low level features in the 21264 can turned on or off. Type
   `make flexible`

3. *Functional debug mode* where a functional simulator checks the correctness of the timing simulator. While running in functional debug mode, early instruction retire should be disabled, and only eio traces, introduced with release 3.0 of the SimpleScalar suite, should be used. Type
   `make functional`

3 Target Specification

In this section we describe the Alpha 21264 microarchitecture, the Compaq DS-10L Alphaserver machine we used as our reference machine, the Digital Continuous Profiling Infrastructure tool from Compaq which allowed us to measure the performance of programs on the native machine, and the microbenchmarks which helped us isolate errors in *sim-alpha*.

3.1 Alpha 21264 Overview

The Compaq Alpha 21264 [2] [3] [8] [9] microprocessor was introduced in 1998. It implements the Alpha architecture, which is a 64-bit load and store RISC architecture. To operate at high clock frequencies, the 21264 incorporates innovative features such as clustered functional units, merging the branch target buffer with the instruction cache, and using a set-predict cache. In the following subsections, we describe the general features of the microprocessor, as well as some of the low level features, which we have implemented in *sim-alpha*.

3.1.1 Microprocessor features

The 21264 has a seven stage pipeline as shown in Figure 1. The fetch stage of the pipeline fetches a set of four instructions from the instruction cache every cycle. It uses the line predictor to get the address of the instruction to fetch the next cycle. The slot stage of the pipeline statically slots instructions to sub-clusters on which they can execute. The branch predictor also returns with a prediction in this stage. The next stage of the pipeline, the map stage, performs renaming of registers and puts instructions in the issue queues. Instructions issue from the integer and floating-point issue queues in the issue stage, read their input operands in the register read stage, and start executing in the functional unit assigned. The instruction outputs are written back in the writeback stage of the pipeline.

Below we list the main features of the 21264. In *sim-alpha*, all these features can be configured with command line parameters, and the default values are those listed in this section.

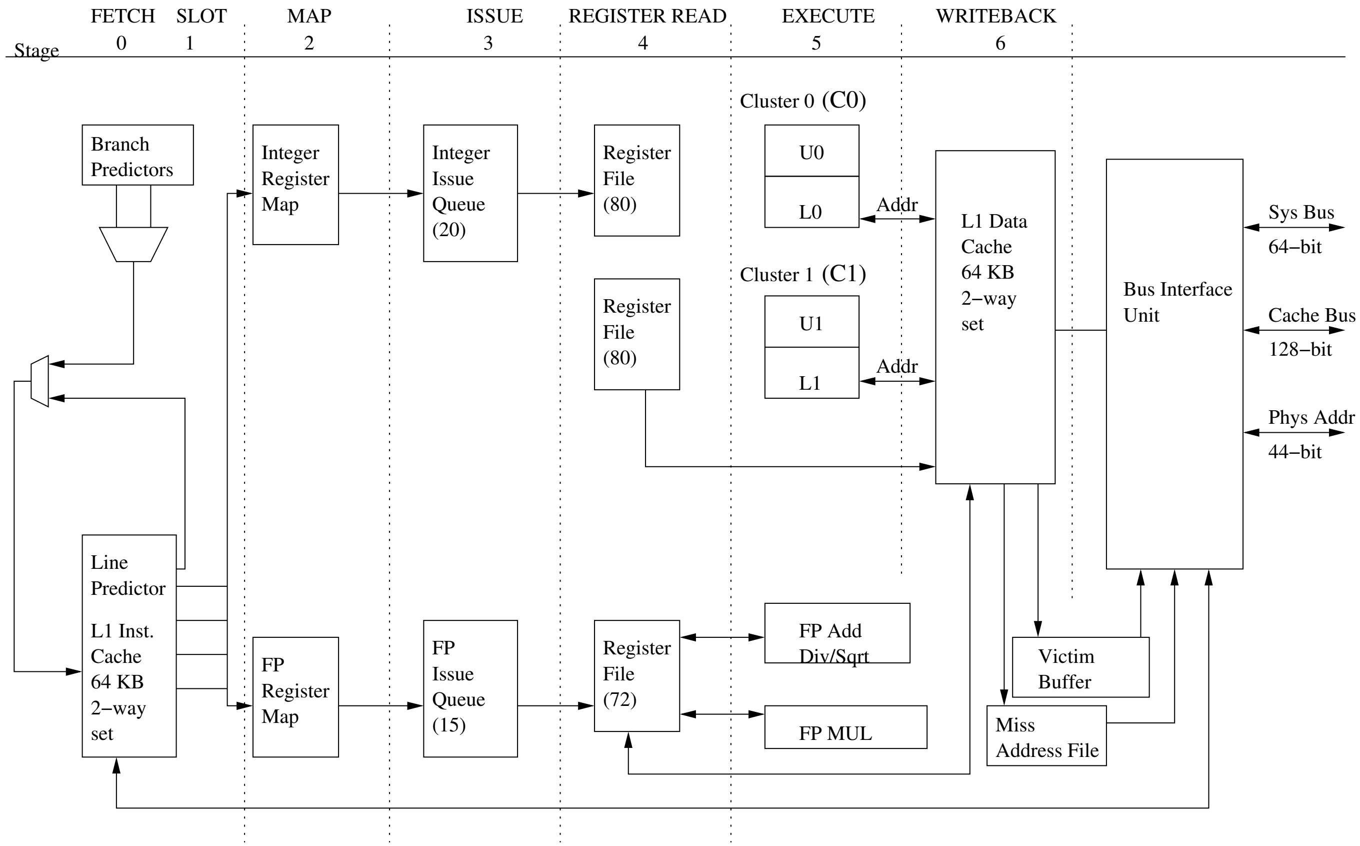

Figure 1: Alpha 21264: Block Diagram (Original diagram courtesy Jim Keller's Alpha 21264 presentation)

- An issue width of six instructions (4 integer and 2 floating point) during each CPU cycle from a 20-entry integer issue queue and a 15-entry floating point issue queue.
- An 80-entry reorder buffer for tracking instructions in flight.
- A demand-paged memory-management unit consisting of a 128-entry, fully-associative data translation buffer (DTB) and a 128-entry, fully-associative instruction translation buffer (ITB).
- Four integer units with an 80 entry register file. These units are called sub-clusters in the alpha, and operate on specific classes of instructions. The 80 entry register file consists of 31 architectural registers, 8 PAL shadow registers, and 41 registers for renaming.
- Two pipelined floating-point units. One unit executes adds, divides, and square roots, and the other unit executes multiplication instructions. The 21264 has 72 floating registers. Of these, 31 are architectural registers, and 41 are used for renaming destination registers of instructions in flight.
- A 64KB virtually addressed instruction cache. The cache is two-way set associative with 64 byte blocks. The 21264 uses a set predictor to choose between the two sets on each access. This ensures single cycle access latency to the I-cache when the set is predicted correctly.
- A virtually indexed, physically tagged dual-read-ported, 64KB data cache. The cache is two-way set associative with 64 byte blocks. The access time for the cache is 3 cycles.
- A tournament branch predictor which consists of
  a) A two level local predictor that has 1024 entries in the first level (indexed by the PC), with 10 bits in each entry, used to index another 1024 entry table of 3-bit saturating counters.
  b) A 4096 entry global predictor with 2-bit saturating counters.
  c) A 4096 entry choice predictor to choose between local and global predictors with 2-bit saturating counters.
- An 8-entry victim data buffer.
- A 32-entry load queue.
- A 32-entry store queue.
- An 8-entry miss address file

3.1.2 Low-level features in the 21264

The following paragraphs describe some of the implementation constraints the designers faced for achieving high clock frequency, and the low-level features they incorporated to achieve high performance.

In the 21264, the branch predictor takes two cycles to make a prediction. This results in a one-cycle bubble between the cycle the instruction is fetched and the cycle the prediction is made. To eliminate this bubble, the 21264 has a line predictor, that effectively acts as a branch target buffer. Each cycle, the line predictor predicts the I-cache line to be accessed in the next cycle. When instructions are fetched from the I-cache, the line prediction bits are also fetched along with the instructions. These bits are used the next cycle to get the next set of instructions. When the branch predictor completes, the prediction is compared with the line predictor in the slot stage of the pipeline. For certain classes of control instructions like branches and immediate jumps, if the branch predictor prediction differs from the line predictor prediction, fetch is re-initiated with the branch predictor address. The line predictor can store a target for a set of four instructions. In *sim-alpha*, using command line parameters, we can vary the number of instructions for which the line predictor stores a prediction. We can also disable the line predictor, and use a regular btb instead.

The instruction cache is two-way set-associative. To achieve single cycle access, a way predictor in the I-cache predicts which set is being accessed in the current cycle. Way prediction gives the effective access time of a direct mapped cache, although it does result in a 2-cycle bubble on a set misprediction. The way predictor latency can be varied in *sim-alpha*.

In the map stage, the processor does not know the number of free registers available to rename in the current cycle. Hence, it ensures that there are always enough registers available to rename for the next two cycles by stalling for 3 cycles, whenever the number of free physical registers falls below 8. After 3 cycles it again evaluates the number of free physical registers, and will stall again for 3 cycles if the free register condition is still unsatisfied.

The integer execution core is partitioned into two clusters C0 and C1. Each cluster has a copy of the 80-entry physical register file, and two sub-clusters called lower (L) and upper (U), containing the integer functional units. These sub-clusters are not symmetric, and contain different numbers and types of functional units. For example, an integer multiply functional unit is present only in U1. The register files contain identical values. These is a one-cycle delay to transfer data from one cluster to another. Thus, dependent instructions can issue during successive cycles only to the same cluster, and will have to wait one cycle to issue to the other cluster. The 21264 statically slots instructions to the two sub-clusters in the slot stage to achieve a better load balance, and then dynamically chooses the cluster during issue. For example, if a fetched octaword contains an `add`, a `mult`, a `load`, and a `shift` instruction in that order, the slot stage will slot it as LULU to ensure maximum usage of execution resources. *sim-alpha* provides command line options for varying the number of clusters, disabling slotting and clustering, and for setting the value of the cross cluster delay.

The D-cache in the 21264 has a 3-cycle hit latency. To facilitate faster instruction wakeup on a cache hit, the 21264 uses a technique called load-use speculation, where it issues instructions dependent on the load assuming a load hit. If the load misses in the cache, these instructions are squashed and reissued. In *sim-alpha*, we approximate load-use speculation by reissuing only the instructions that are dependent on the missing load.

The 21264 also uses prefetching on an I-cache miss to improve performance. The 21264 can prefetch four instruction cache lines from the L2 cache on an I-cache miss. Four lines is also the default prefetch value in *sim-alpha*. However, the number of lines to prefetch can be varied using command line parameters. The 21264 has an 8-entry unified victim buffer to cache recently evicted blocks from the I-cache, D-cache, and L2 cache. *sim-alpha* caches blocks only from the level-one D-cache in the victim buffer. The size of the victim buffer can be varied in *sim-alpha*.

The branch predictor uses an adder to precompute targets of immediate branches. This adder enables the 21264 to predict targets of immediate branches correctly even if the line predictor is wrong. The 21264 also has a mechanism called early instruction retire to detect no-ops early in the pipeline (map stage). These no-ops are retired immediately, and thus do not consume execution resources. The user can enable or disable the adder and the early instruction retire in *sim-alpha*.

To enforce correct memory accesses, the 21264 uses order traps. Order traps result in the pipeline being flushed, and the instruction being restarted from the fetch stage of the pipeline. There are two main types of order traps: Load-Load order traps and Store-Load order traps. The 21264 invokes a load trap on a newer load instruction that has been issued before an older load instruction to the same address. To detect a store trap, the 21264 compares the addresses of all store instructions as they are issued to loads in the load queue. If the processor detects a newer load to the same address in the load queue, it invokes a store trap on the newer load. Store traps are necessary to ensure that loads and stores to the same address happen in program order. Traps are expensive in terms of performance, the minimum cost being 12 cycles. Hence the 21264 uses special hardware to reduce the occurrence of store traps. The processor has a 1024 one bit table called the *stWait* table, indexed by the PC, to stall issue of loads causing order traps. This bit is fetched from the I-cache with each instruction. The processor does not issue a load for which the *stWait* bit is set, until all previous stores have issued. On a store trap, this bit is set for the faulting load when it is re-fetched. All bits in the *stWait* table are unconditionally cleared every 16,384 cycles. The 21264 also has another type of trap called Mbox trap, for ensuring correctness in the memory system. The Mbox traps also result in a flushing of the pipeline but are triggered by events occurring in the memory system such as outstanding misses to two loads to same address but different destination registers, outstanding misses to different physical addresses that map to the same D-cache or L2 cache line, and store queue overflow. The Load-Load order traps and the Mbox traps can be disabled in *sim-alpha*. The user can also set the size of the *stWait* table.

3.2 Compaq DS-10L Alphaserver

We used the Compaq DS-10L Alphaserver to validate *sim-alpha*. The workstation has a single 21264 processor clocked at 466 MHz, a 2 MB external L2 cache (direct mapped, with 64 byte blocks), and 256 MB of physical memory. The workstation runs version 5.1 of Compaq Tru64 UNIX. The DS-10L has custom memory controller chips which consists of a single control chip, the Digital DC1046C, and two chips which act as data switches, the Digital DC1047B. The SDRAM consists of 16 chips of 8MB each running at 125 MHz and an 8-ns access time. The C compiler on the DS-10L is version 6.3-025 of the Compaq C compiler.

3.3 Digital Continuous Profiling Infrastructure

The Digital Continuous Profiling Infrastructure (DCPI) [4] for Compaq Alpha platforms permits continuous low-overhead profiling of entire systems, including the kernel, user programs, drivers, and shared libraries. DCPI (subsequently renamed Continuous Profiling Infrastructure) samples the Alpha performance monitoring counters to collect information about each program running on the system.

DCPI can be used for measuring the frequency of certain events on the Alpha 21064, 21164, and the 21264. On the 21264, DCPI can measure the number of cycles taken by a program, number of instructions retired, Mbox traps incurred, number of retired itlb misses, number of single and double dtlb misses, number of retired conditional branches, and number of retired unaligned traps

DCPI calculated the number of cycles taken by the programs to complete on the native DS-10L system. This number can then be compared against the number of cycles taken in the simulator, to compute the simulator error.

3.4 Microbenchmarks

Figure 2 gives a brief description of the microbenchmarks we used for validating *sim-alpha*. The first row lists the microbenchmarks we used for testing the front-end such as the line predictor implementation and the branch predictor implementation. The C infront of the names of these microbenchmarks signifies that these test control flow. The second row lists the microbenchmarks for testing the execution core such as the scheduler. The E in the microbenchmark name stands for the execution core. The last row lists the microbenchmarks for testing the memory system parameters such as the level-one cache latency, the level-two cache latency, and the main memory latency. The M here stands for the memory system.A more complete description of the microbenchmarks can be found in [6] [7]. The source code for all the microbenchmarks can be obtained from the website listed in Section 2.

C-C:

```
/* if-then-else benchmark which repeatedly toggles between the if block and the else block. C-Ca and C-Cb represent two different assembly versions of C-C */
for (i=0; i<1000000; i++) {
    j = i % 2;
    if (!j)
        p++;
    else
        r++;
}
```

C-Ca:

```
and   t0,0x1,t2
blt   t0, <144:loop_end>
unop
unop
bne   t2, <80:else>
addl  a1,0x1,a1
br         <84:endif>
unop
```

C-Cb:

```
and   t0,0x1,t2
blt   t0, <144:loop_end>
bne   t2, <80:else>
addl  a1,0x1,a1
br         <84:endif>
unop
```

C-R:

```
/* A recursive bechmark which recurs 1000 levels deep */
static int m=0;

void func(int k, int j) {
  m = j + k;
  if (!k) return;
  else
    func(k-1, j);
}
```

C-Sn:

```
/* Switch benchmark to test the line predictor. Contains indirect jumps which are taken 1,2, and 3 times */
for (i=0; i<4000000; i++) {
    j = i % 10;
    switch (j) {
        case 0,1,2:
          k++;
          break;
        case 3,4,5:
          l++;
          break;
    /* repeat case 8 times */
    }
}
```

E-I:

```
/* Series of independent arithmetic integerops operations */
int k, l, m, n, o, p, q, r;

for (i=0; i<250000; i++) {
    k = k + i;
    l = l + i;
    m = m + i;
    n = n + i;
    o = o + i;
    p = p + i;
    q = q + i;
    r = r + i;
    /* repeat 20 times */
}
```

E-F:

```
/* Series of independent arithmetic fp operations */
float k, l, m, n, o, p, q, r;

for (i=0; i<250000; i++) {
    k = k + i;
    l = l + i;
    m = m + i;
    n = n + i;
    o = o + i;
    p = p + i;
    q = q + i;
    r = r + i;
    /* repeat 20 times */
}
```

E-Dn:

```
/* String of depenedent operations */
for (i=0; i<250000; i++) {
    a = d + i;
    b = a + i;
    c = b + i;
    d = c + i;

    a = d + i;
    b = a + i;
    c = b + i;
    d = c + i;

/* repeat n insts 320/n times */
}
```

M-I:

```
/* Series of independet loads to L1 data cache */
for (i=0; i<8192; i++)
    a[i] = i;

for (r=0; r<1500; r++)
    for (j=0; i<8191; i++)
        j = a[i]+j;
```

M-D:

```
/* Series of dependent loads to L1 data cache */
for (i=0; i<2047; i++)
    a[i] = (int *)&a[i+1];

for (i=0; i<15000; i++) {
    b[i] = (int **)a[0];
    for (k=1; k<1000;  k++)
        b = (int **) b[1];
}
```

M-L2:

```
/* Series of dependent loads to L2 cache */
for (i=0; i<131071; i++) {
    a[i] = (int *) &a[i+1]
}
for (i=0; i<1500; i++) {
    b = (int **)a[7];
     for (k=8; k<13105; k+=8) {
        b = (int **) b[8];
     }
}
```

M-M:

```
/* Series of dependent loads to main memory */
for (i=0; i<524387; i++) {
    a[i] = (int *)&a[i+1];
}
for (i=0; i<1500; i++) {
    b = (int **) a[7];
    for (k=8; k<13105; k+=8)
        b = (int **) b[8];
    b = (int **) a[7+262144];
    for (k=8; k<13105; k+=8)
        b = (int **) b[8];
}
```

Figure 2: Microbenchmarks

Table 1: Microbenchmark Error

benchmark	21264 IPC	sim-alpha IPC	%error
C-Ca	1.80	1.87	4.3
C-Cb	1.87	0.87	0.6
C-R	2.65	2.66	0.2
C-S1	0.56	0.54	-4.3
C-S2	0.85	0.87	2.5
C-S3	0.95	0.95	0.3
C-C0	1.75	1.68	-4.2
E-I	4.00	3.97	-0.9
E-F	1.01	1.01	0.2
E-D1	1.03	1.04	0.4
E-D2	2.16	2.01	-7.2
E-D3	2.72	2.71	-0.3
E-D4	2.79	2.90	3.7
E-D5	3.3	3.23	-1.9
E-D6	3.11	3.15	1.3
E-DM1	0.15	0.15	-0.3
M-I	2.98	2.99	0.6
M-D	1.66	1.66	0.3
I-P	1.75	1.83	4.3
M-L2	0.29	0.28	-2.3
M-M	0.06	0.06	-1.8
Mean Absolute Error			2.0

4 Simulator Performance

In this section, we tabulate the error of *sim-alpha* across our set of microbenchmarks, and a set of 22 macrobenchmarks from the SPECCPU 2000 suite. The error was computed using the formula

$$\%\,Error \;=\; \frac{(Cycles\ on\ the\ native\ machine\ -\ Cycles\ on\ the\ simulator)\ *\ 100}{(Cycles\ on\ the\ native\ machine)}$$

4.1 Microbenchmark Evaluation

Table 1 shows the results of *sim-alpha* across our set of microbenchmarks. We can see from Table 1 that the error is less than 5% for all but E-D2. The mean absolute error across our set of microbenchmarks is 2%.

4.2 Macrobenchmark Evaluation

Tables 2 and 3 show the error for *sim-alpha* across the SPECCPU 2000 integer and floating-point benchmarks. From Table 2 we can see that the mean absolute error of *sim-alpha*, across the integer benchmarks, is 6.6%. Looking at Table 3 however, we see that *sim-alpha* shows greater error across the set of floating point benchmarks. We attribute insufficient modeling of the floating point

Table 2: SPECINT 2000 Error

benchmark	21264 IPC	sim-alpha IPC	%error
gzip	1.53	1.59	1.71
vpr	1.02	1.06	2.77
gcc	1.04	1.17	9.59
mcf	0.60	0.64	15.65
crafty	1.40	1.37	0.68
parser	1.18	1.34	11.30
eon	1.21	1.21	-1.21
gap	0.87	0.97	8.58
bzip2	1.74	1.55	-8.58
twolf	1.10	1.07	-6.30
Mean Absolute Error			6.64

Table 3: SPECFP 2000 Error

benchmark	21264 IPC	sim-alpha IPC	%error
wupwise	1.58	1.39	-7.47
swim	0.87	0.79	-11.02
mgrid	1.32	0.95	-37.23
applu	1.09	1.15	6.41
mesa	1.57	1.17	-38.37
galgel	1.45	1.91	24.55
art	0.48	0.85	43.64
equake	1.02	0.84	-24.46
facerec	1.14	1.10	5.99
ammp	0.47	0.55	23.04
lucas	1.57	1.40	-12.32
apsi	0.79	0.93	15.81
Mean Absolute Error			21.52

pipeline, and inaccuracies in our memory system implementation to be the source for the error in the floating point benchmarks [6] [7].

5 Using *sim-alpha*

sim-alpha supports many of the options found in *sim-outorder*. It also has a number of new options to support the 21264 pipeline, and some options found in *sim-outorder* are not included. The source code includes a number of new files for modeling the 21264 microarchitecture. In this section, we describe the various command line options in *sim-alpha*, and also describe its internal workings.

5.1 Command Line Options

5.1.1 Processor core configuration

All the options in this subsection take an integer argument unless otherwise specified

-mach:freq - Frequency of simulated machine. This is used by system calls like *rusage* to return *time* values consistent with that on a native machine with same frequency.

-fetch:ifqsize - Fetch queue size. The fetch stage stalls if the fetch queue cannot accommodate fetch width number of instructions.

-fetch:speed - Number of discontinuous fetch width fetches per cycle.

-fetch:width - Number of instructions to fetch each cycle.

-slot:width - Number of instructions which can be slotted per cycle.

-map:width - Number of instructions which can be mapped per cycle.

-issue:intwidth - Number of integer instructions which can be issued per cycle.

-issue:fpwidth - Number of floating point instructions which can be issued per cycle.

-commit:width - Number of instructions which can be committed per cycle.

-fetch:stwait - Specifies size of the stWait table.

-issue:int_reg_lat - Integer register read latency.

-issue:fp_reg_lat - Floating point register read latency.

-issue:int_size - Integer issue queue size.

-issue:fp_size - Floating point issue queue size.

-reg:int_p_regs - Number of integer physical registers.

-reg:fp_p_regs - Number of floating point physical registers.

-rbuf:size - Specifies the size of the reorder buffer.

-lq:size - Specifies the size of the load queue.

-sq:size - Specifies the size of the store queue.

-res:iclus - Number of integer execution clusters in the processor core.

-res:fpclus - Number of floating point clusters in the processor core.

-res:delay - Minimum cross cluster delay between adjacent clusters.

5.1.2 Memory Hierarchy flags

-bus:queuing_delay - Enables delay due to contention in the buses to lower levels of the memory hierarchy.

-cache:perfectl2 - Simulates perfect level 2 cache.

-prefetch:dist - Number of blocks to prefetch on a level 1 I-cache miss .

-cache:define - *<name>*:*<nsets>*:*<bsize>*:*<subblock>*:*<asso>*:*<repl>*:*<lat>*:*<trans>*:*<#resources>*:*<rescode>*

where

<name> - Defines name of the cache.
<nsets> - Specifies the number of sets in the cache.
<bsize> - Specifies the block size of the cache.
<subblock> - Specifies sub-block size of the cache.
<asso> - Specifies associativity of the cache.
<repl> - Specifies the cache replacement policy. Choose from LRU, FIFO, random, and LFU.
<lat> - Hit latency.
<trans> - Translation policy (vivt, vipt, pipt).

-cache:vbuf_lat - Victim buffer latency.

-cache:vbuf_ent - Number of entries in the victim buffer.

-cache:mshrs - Sets maximum number of MSHRs per cache.

-cache:prefetch_mshrs - Sets maximum number of prefetch MSHRs per cache.

-cache:mshr_targets - Sets maximum number of allowable targets per mshr.

-bus:define - *<name>*:*<width>*:*<cyclelatency>*:*<arbitrationpenalty>*:*<infbandwidth>*:*<#resources>*:*<resourcecode>*:*<resourcenames>*

where

<name> - Specifies bus name.
<width> - Specifies bus width in bytes.
<cyclelatency> - bus latency in terms of processor cycles.
<arbitrationpenalty> - Number of cycles for arbitration.
<infbandwidth> - if 1, infinite bandwidth.
<resourcenames> - Name of structure to which this bus connects (eg. L2).

-mem:clock_multiplier - ratio of cpu frequency to DRAM frequency.

-mem:page_policy - DRAM page policy (O - open page, 1 - close page autoprecharge).

-mem:ras_delay - time between start of ras command and cas command.

-mem:cas_delay - time between start of cas command and data start.

-mem:pre_delay - time between start of precharge command and ras command.

-mem:data_rate - 1 specifies single data rate and 2 specifies double data rate.

-mem:bus_width - width of bus from cpu to dram.

-mem:chipset_delay_req - delay in chipset for request path.

-mem:chipset_delay_return -delay in chipset in data return path.

-tlb:define - *<name>*:*<nsets>*:*<bsize>*:*<subblock>*:*<assoc>*:*<repl>*:*<hitlatency>*:*<translation>*:*<prefetch>*:*<#resources>*:*<resourcecode>*
the flags have similar meanings to cache define.

5.1.3 Predictor configuration (Line and branch predictor)

-line_pred:ini_value - Initial value of the 2-bit line predictor training counter.

-line_pred:width - Number of instructions for which the line predictor stores a prediction.

-way:pred - Specifies the latency of the way predictor.

The branch predictor configuration supports the following new flags along with those specified in [1].

-bpred 21264 21264 tournament predictor

The predictor-specific arguments are listed below:

-bpred:21264 - <*l1size*> <*l2size*> <*lhist_size*> <*gsize*> <*ghist_size*> <*csize*> <*chist_size*>

where the different flags are :
<*l1size*> - the local predictor level 1 table size.
<*l2size* > - the local predictor level 2 table size.
<*lhist_hist*> - the number of history bits in the second level table.
<*gsize*> - the size of global predictor.
<*ghist_size*> - number of history bits in the global predictor.
<*csize*> - choice predictor size.
<*chist_size*> - number of history bits in the choice predictor.

5.2 Low-level feature configuration

For each of these features, a 1 enables it and a 0 disables it. These features can be made operational by compiling the simulator with the FLEXIBLE_SIM flag.

-issue:no_slot_clus - 1 disables clustering of functional units.

-slot:adder - Enable/disable the adder for computing targets of PC relative control instructions in the slot stage.

-slot:slotting - Enable/disable static slotting in the slot stage.

-map:early_retire - Enable/disable early instruction retire in the map stage.

-wb:load_trap - Enable/disable load traps.

-wb:diffsize_trap - Enable/disable traps due to loads and stores with different sizes to the same address.

-cache:target_trap - Enable/disable trap if two loads map to same MSHR target.

-cache:mshrfull_trap - Enable/disable trap if MSHRs are full.

-map:stall - Enable/disable stall of map stage for 3 cycles if the number of free physical registers to map falls below 8.

-bpred:spec_update - Enables speculative update of the global and choice predictors if 1.

-line_pred:spec_update - Enables speculative update of line predictor if 1.

-load:spec - Enable/disable load use speculation.

5.3 *sim-alpha* internals

This section, describes the internal working of *sim-alpha*. Since *sim-alpha* is an execution-driven simulator, it executes instructions down the mis-speculated path, in the same way an actual processor would execute them. Thus, *sim-alpha* can capture the behavior of these mis-speculated instructions. However, a disadvantage of this approach is that the correct path is known only at commit time, and hence unlike *sim-outorder* perfect prediction cannot be simulated easily. Each stage of the 21264 pipeline is modeled in a separate file, except the register read stage, which is bundled along with the issue stage. The main loop of the simulator, located in `simulate.c`, appears as follows:

```
while(TRUE) {
  eventq_service_events();
  commit_stage();
  writeback_stage();
  issue_stage();
  map_stage();
  slot_stage();
  fetch_stage();
}
```

This loop is walked once for each simulated machine cycle. The simulation is partly event driven. Before the entering the main loop, the simulator calls the initialization functions for each pipeline stage. These functions set up the various data structures needed for timing simulation. Like *sim-outorder*, if fast-forwarding is enabled, functional simulation is performed till the required number of instructions are committed, and then timing simulation starts.

The fetch stage of the pipeline is implemented in *fetch_stage()* and is in the file `fetch.c`. The fetch stage accesses the instruction cache, and fetches fetch_width number of instructions per access. The accesses are always aligned on a fetch_width boundary. The fetch speed determines how many discontinuous fetches can be performed by the processor per cycle. The line predictor provides the next I-cache line to fetch.

The slot stage of the pipeline resides in `slot.c` and is implemented by the function *slot_stage()*. In the slot stage, instructions are statically slotted to either the UPPER or the LOWER sub-clusters depending on their position in the fetch packet, and their resource requirements. The slotting algorithm is implemented as an array *slot_instclass2slotclass*. The control instructions also access the branch predictor in this function with a call to *bpred_lookup()*. For PC-relative control instructions, the target is computed using an adder. The speculative update of the branch and the line predictors also happens in this stage. For certain classes of instructions, the slot stage overrides the line predictor prediction with the branch predictor prediction if they differ. In this case, the remaining instructions in the fetch queue are squashed, and fetch is restarted the next cycle by changing *regs_PC* to the branch predictor target address. We also check to make sure that the fetch is continuous in the absence of control instructions. If not, the fetch is restarted with the correct PC. However, we don't charge any penalty here as the 21264 fetch engine can detect the absence of control instructions the same cycle.

The map stage is implemented in the function *map_stage()* in `map.c`. The map stage initially identifies the input and output registers for the current instruction. It then checks for the availability of a reorder buffer entry, integer or floating point issue queue entry (depending on whether the instruction has an integer or floating point destination register), output physical register, and load or store queue entry if the instructions is a load or store instruction. The map stage then identifies the input physical physical registers for this instruction, and if these are ready, places the instruction in the ready queue. Otherwise, the map stage identifies the producer(s) of the instruction's operands, and queues the instruction for wakeup, by linking it to the dependence chain entry of the producer's reorder buffer entry. Some instructions with destination register 31 are also retired in this stage as part of early instruction retire. FTOI and floating point store instructions are queued in both integer and floating point queues. ITOF and floating point load instructions are queued in the integer queue. The map stage also checks the stWait bit for load instructions, and if set, it adds these instructions to the dependence chain of the last store instruction before the current load. CMOV instructions have three input operands, and these are handled using the *cmovdeps* dependence chain. The map stage also assigns a unique *inum* to each instruction, which is used for determining its age, for insertion into the ready queue.

The code for the issue stage resides in the file `issue.c`, and is implemented in the function *issue_stage()*. The issue stage is responsible for picking instructions from the integer and floating point ready queues, checking for the availability of functional units, and issuing the instruction to the functional unit. The register read latency is also charged here. Instructions whose operands are ready are inserted in the integer and floating point ready queues by the *writeback_stage()* and the *map_stage()*. These instructions also have corresponding entries in the integer and floating point issue queues. Instructions are ordered in the ready queue strictly by age. Each cycle, the issue stage issues *issue width* number of oldest instructions from the ready queues. The issue stage uses the slotting performed by the slot stage to decide which sub-cluster an instruction should be issued to. The functional unit code is contained in `resource.c`. The function *res_find_clus()* finds the cluster on which an instruction can execute. The functional unit allocation is less flexible in *sim-alpha* than other simulators because of the presence of clusters and sub-clusters in the 21264. By removing clustering, the functional unit code can be made more scalable. The issue stage also assigns the appropriate latency for each instruction, depending on its type. The issue stage finally sets up events to free the instruction entry in the integer and floating point queues 2 cycles after issue, and to signal completion of the execution of this instruction.

The writeback stage implemented in *writeback_stage()* is responsible for waking up the dependent instructions when a producing instruction completes. This function is contained in the file `writeback.c`. In the simulator, the functional execution of the instruction also takes place in this stage. The writeback stage picks up completed instructions from the event queue, and walks their dependence chain marking the pertinent operands of dependent instructions ready. If all the operands of an instructions are ready, the routine inserts this instruction into the ready queue. The writeback stage also sets the completed flag in the reorder buffer entry for these instructions. The targets for control instructions are resolved, and mis-predictions are indicated in the corresponding reorder buffer entry, along with the correct target. Load instructions access the D-cache, and store instructions are marked as ready to access the D-cache during commit.

The commit stage in *commit_stage()* is responsible for retiring instructions from the reorder buffer. The `commit.c` contains the code for this function. Every cycle, the commit stage retires *commit_width* number of instructions. As in the 21264, the commit stage does not retire past a branch in one cycle. The commit stage examines the head of the reorder for mis-predictions and traps, and flushes the pipeline by calling *commit_flush_pipeline()* if the instruction caused a mis-predict or a trap. Otherwise, it retires the instruction, and updates the architectural register file. The commit stage also sends the store instructions to the D-cache.

sim-alpha incorporates a detailed memory system with support for a multi-level cache hierarchy, address translation, bus contention, and an SDRAM memory model. The memory code is mostly event driven, and is scheduled by the global event queue contained in `eventq.c`. At the beginning of each cycle, the *eventq_service_events()* function checks for memory operations completing that cycle. The files `cache_timing.c, tlb.c, bus.c, memory.c,` and `dram.c` contain most of the memory timing code for *sim-alpha*.

6 Summary

sim-alpha provides a flexible, validated baseline for researchers to evaluate new architectural enhancements. By incorporating many of the constraints found in a real implementation, *sim-alpha* provides a lower-level point of comparison to measure trends. The option to turn off these constraints, provides the researcher with the ability to remove the features which might interfere with the evaluation of new higher-level ideas, while retaining the low level features which are orthogonal to the idea being evaluated. Future work on *sim-alpha* may include reducing floating point benchmark errors and building a more flexible functional unit model. Future work may also include building a more accurate memory system, and measuring error using scientific and multimedia workloads.

References

[1] D. Burger and T. M. Austin. The simplescalar tool set version 2.0. Technical Report 1342, Department of Computer Sciences, University of Wisconsin-Madison, June 1997.

[2] Compaq Computer Corporation. *Alpha 21264 Microprocessor Hardware Reference Manual*, July 1999.

[3] Compaq Computer Corporation. *Compiler Writer's Guide for the Alpha 21264*, 1999.

[4] Compaq Systems Research Center. *Digital Continuous Profiling Infrastructure project. http://www.research.digital.com/SRC/dcpi.*

[5] Compiler and H. P. Architecture Research Group. Trimaran, an infrastructure for research in instruction-level parallelism. http://www.trimaran.org/.

[6] R. Desikan. Design and validation of an accurate microprocessor simulator. Master's thesis, Department of Electrical and Computer Engineering, The University of Texas at Austin, December 2001.

[7] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 266–277, June 2001.

[8] R. Kessler. The alpha 21264 microprocessor. *IEEE micro*, 19(2):24–36, Mar 1999.

[9] R. Kessler, E. McLellan, and D. Webb. The alpha 21264 microprocessor architecture. In *Proceedings of International Conference on Computer Design*, pages 90–105, October 1998.

[10] V. Pai, P. Ranganathan, and S. Adve. Rsim: A simulator for shared-memory multiprocessor and uniprocessor systems that exploit ilp. In *Proceedings of the 3rd Workshop on Computer Architecture Education*, 1997.

[11] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer simulation: The SimOS approach. In *IEEE Parallel and Distributed Technology*, 1995.