WaveScript Benchmarks Performance Report

November 6, 2008

Machine information:
Linux chastity 2.6.22-14-generic #1 SMP Tue Feb 12 07:42:25 UTC 2008 i686 GNU/Linux
LD_PRELOAD: CC: cc

WaveScript SVN:
Revision: 3631

WaveScope Engine SVN:
(omitted for now)

1 Microbenchmarks

This section reports various microbenchmarks that stress the implementation of particular language constructs or data types.
Per-stream-element overheads

One thing that you can see, is that currently (2007.10) the C++/XStream engine has a high per-tuple (that is, per-element) on the communication channels relative to the ML backend. The just_timer test stresses this, doing nothing but passing a large number of unit tuples.

Focusing on scheduling overheads a bit more, we turn to the following data passing microbenchmarks. These do nothing but generate a stream of numbers, and then add up windows of those numbers. We vary the window size in the following graphs. The numbers are passed either one at a time (“raw”), or in bulk using arrays or lists.

Notes:
• FFT results for Scheme above depend on whether or not it is configured to use FFTW, or a native Scheme fourier transform.

2 Language Shootout Benchmarks

This is where I will accumulate some of the small benchmarks from the language shootout. Here are some per-benchmark comments:

• fannkuch - “pancake flipping”. This is a translation of the gcc version of the benchmark. Tests indexed access to a small array.

3 Application Benchmarks

This section includes performance results on larger programs, namely, our current applications.

3.1 Marmot Application

We start off by looking at the original, hand-optimized marmot application that we deployed. We break it down by phase: the first three phases of the computation, followed by all three together.
3.2 Computer Vision: Background Subtraction

4 Data Representation Profiling

This is stale data for now... having sneaky problems with the datarep Makefile that are hosing regression tests. [2007.11.07]

This section includes an analysis of the efficiency of different data representations under different backends. This should theoretically be run on different hardware platforms as well (such as the ARM-based ensboxes).
4.1 Arrays of Arrays

Arrays of arrays are notable because they cannot generally be flattened (the inner arrays will always be pointers). In the future we may look at tentative flattening based on profiling data. But first, here are the times for repeatedly allocating an array of arrays, and for repeatedly folding the values in an array of arrays.

Next we look at allocating arrays of tuples and vice versa. We look at both square sizes and at highly skewed dimensions. This is limited by not being able to make tuples very large.

Then we do examine folding over arrays of tuples and tuples of arrays.
A Appendix: Raw numbers for above graphs

Microbenchmarks

## Real or User time for each benchmark/backend
## LD_PRELOAD:
## NOSUGO:
## NICE:
## CC:
## ‘which icc’ =
Benchmark mlton03 c2boehm c2boehmseglist c2 c2seglist c2def c2defseglist
just_timer 2496.000 2524.000 2540.000 2504.000 2524.000 5040.000 5028.000
readfilebigwins 3708.000 508.000 1228.000 1292.000 3764.000 276.000 868.000
printing_lists 2692.000 920.000 920.000 868.000 864.000 820.000 820.000
conv_SigsegArr 2600.000 372.000 7240.000 804.000 5844.000 52.000 6492.000
fft 128.000 948.000 924.000 908.000 956.000 792.000 864.000

Language Shootout:

## Real or User time for each benchmark/backend
## LD_PRELOAD:
## NOSUGO:
## NICE:
## CC:
## ‘which icc’ =
Benchmark c2
fannkuch2 4920.000

Application Benchmarks:

## Real or User time for each benchmark/backend
## LD_PRELOAD:
## NOSUGO:
## NICE:
## CC:
## ‘which icc’ =
Benchmark mlton03 c2boehm c2boehmseglist c2 c2seglist c2def c2defseglist
# Running benchmark marmot1.bench for 100 tuples.
run_first_phase 7292.000 11821.000 4088.000 7380.000 5640.000 7344.000 4052.000
# Running benchmark marmot2.bench for 150 tuples.
test_marmot2 2196.000 5232.000 5256.000 4644.000 4752.000 4556.000 4580.000
# Running benchmark marmot3.bench for 14 tuples.
test_heatmap 7748.000 3244.000 3244.000 2508.000 2496.000 3220.000 3216.000
B Appendix: Additional system information

Top results before running benchmarks:

```
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
1 root 21 0 2948 1856 532 S 0 0.1 0:08.33 init
2 root 11 -5 0 0 0 S 0 0.0 0:00.00 kthread
3 root RT -5 0 0 0 S 0 0.0 0:05.69 migration/0
4 root 34 19 0 0 0 S 0 0.0 0:14.78 ksoftirqd/0
5 root RT -5 0 0 0 S 0 0.0 0:00.00 watchd/0
6 root RT -5 0 0 0 S 0 0.0 0:04.87 migration/1
7 root 34 19 0 0 0 S 0 0.0 0:04.00 ksoftirqd/1
8 root RT -5 0 0 0 S 0 0.0 0:00.00 watchd/1
9 root 10 -5 0 0 0 S 0 0.0 0:01.78 events/0
10 root 10 -5 0 0 0 S 0 0.0 0:00.04 events/1
11 root 10 -5 0 0 0 S 0 0.0 0:00.02 khelper
31 root 10 -5 0 0 0 S 0 0.0 0:01.76 kblockd/0
32 root 10 -5 0 0 0 S 0 0.0 0:00.04 kblockd/1
```

Top results after running benchmarks:

```
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
1 root 15 0 2948 1856 532 S 0 0.1 0:08.33 init
2 root 11 -5 0 0 0 S 0 0.0 0:00.00 kthread
3 root RT -5 0 0 0 S 0 0.0 0:05.69 migration/0
4 root 34 19 0 0 0 S 0 0.0 0:14.78 ksoftirqd/0
5 root RT -5 0 0 0 S 0 0.0 0:00.00 watchd/0
6 root RT -5 0 0 0 S 0 0.0 0:04.87 migration/1
7 root 34 19 0 0 0 S 0 0.0 0:04.00 ksoftirqd/1
8 root RT -5 0 0 0 S 0 0.0 0:00.00 watchd/1
9 root 10 -5 0 0 0 S 0 0.0 0:01.78 events/0
10 root 10 -5 0 0 0 S 0 0.0 0:00.04 events/1
11 root 10 -5 0 0 0 S 0 0.0 0:00.02 khelper
31 root 10 -5 0 0 0 S 0 0.0 0:01.76 kblockd/0
32 root 10 -5 0 0 0 S 0 0.0 0:00.04 kblockd/1
```