The performance of select STAMP benchmarks with transactional cache hardware configurations

Andy Bartholomew

December 19, 2008

1 Introduction

I have adapted STAMP benchmarks [1] for use with the MPARM simulator in order to evaluate the performance of transactional cache configurations. We are primarily interested in evaluating energy consumption in multi-core mobile devices.

There are two main hardware configurations being evaluated: a simple transactional cache ("vanilla TM"), and a victim cache configuration. In the vanilla configuration all transactional data is stored in the transactional cache. In the victim cache configuration, transactional data is kept in local caches, but evicted transactional cache lines are stored in this victim cache. In general, this allows for larger read/write sets without overflowing, but increases the chance of aborts. There are several policy variations on these models that control the shutting down of the extra cache when not in use. However, The benchmarks I focused on spend almost all of their time executing transactions, so these models were not considered.

I had previously adapted the *kmeans* benchmark, but concluded that the read/write set was too small and the transaction length too short to demonstrate the utility of a transactional cache. For this reason I next chose the two benchmarks in the STAMP suite that have medium-sized read/write sets, *vacation* and *genome*. In general, these benchmarks frequently overflow the simple transactional cache, and make consistent use of the victim cache.

In the following sections I detail the non-trivial methods used to adapt the STAMP benchmarks, discuss the structure of the two benchmarks, and outline some results.

2 Methods

This section serves primarily as documentation for future project members.

2.1 Initialization

In the original benchmarks, initialization was done before a special parallel function call with a single pointer to all allocated data structures. In our simulations, any work done before threads are spawned is done by the primary thread while the others wait for initialization. Pointers to all the structures allocated during this initialization phase may be passed using the `make_global_point` and `get_global_point` functions provided in `appsupport.h`

2.2 `init_multi`

The original benchmarks made use of system calls not available in the MPARM simulator. In particular, MPARM does not provide printf, malloc, or pthread support. While there is an analogous printing function, and various processor synchronization mechanisms, we had to implement our own version of malloc for the simulator.

The original shared memory allocation implementation proved to be too contentious, so I implemented a version that partitions the shared memory space and uses separate metadata structures for each core. However, because the primary core is reponsible for making the vast majority of allocations during the initialization phase, it is best to partition the remaining memory and initialize all metadata at the end of initialization. This is done by calling `init_multi`. Note that this is only necessary of the benchmark makes allocations after the initialization phase. In the future it may be beneficial to make the call to `init_multi` automatic.

Once `init_multi` has been called, each core allocates memory from its own partition. A call to `sfree_multi`, however, may point to a block in any partition. In this case we logically mark the block as free without moving it

from the metadata list, which is owned by another core. Functions for lazy cleanup have been implemented, but have not yet been integrated into the system, as they have not proven necessary.

2.3 Generating parameters and input

Because MPARM does not provide file input or command line arguments, I have created python scripts that generate header files with the parameters or input defined by macros. These scripts are called `geninput.py` or `genparams.py`, and their arguments reflect the arguments of the original benchmarks.

2.4 `stamp_suite.py`

I have converted `launch_suite` from bash to python, which I find easier to read and maintain. This script is specific to the stamp benchmarks, which all have a run[name] script contained in their application folders and take the same parameters. I have also converted the data collection scripts to python.

3 Benchmarks

Vacation is essentially a set of red-black tree databases. The user specifies some number of tasks to be performed on these databases, with a specified number of tasks per transaction. These tasks consist of reservations (look-ups), customer removal (deletes), and insertions. The user specifies the proportion of look-ups to be performed, with higher proportions causing less contention. We have found that even one lookup on a fairly small database causes an overflow for the vanilla configuration.

Genome is primarily a hashmap, followed by a parallelized Rabin-Karp string search algorithm. There are two phases: in the first phase, elements are added to the hashmap, six per transaction. In the second phase, we make matching attempts on the sequence inside a transaction.

4 Results

4.1 Vacation

With minimal contention, the victim cache configuration scales much better than the vanilla configuration. In fact, it is surprising that the vanilla configuration has any speedup at all, considering it has a 100% overflow rate. Also of interest is that as cores are added, the victim cache configuration uses more energy than the vanilla configuration. The majority of this difference comes from the CPUs, possibly because the VC takes more energy waiting to use the bus.

With a small amount of contention things really aren't making much sense. Neither configuration gets any speedup from increasing the number of cores. This indicates that aborts are particularly bad with this benchmark, to an extent beyond what we should expect.

4.2 Genome

The vanilla configuration performs the fastest for genome, although it does not gain much speedup as cores increase. Meanwhile the VC configuration performs very badly with one core, but quickly improves as we increase to 4 cores. Between 2 and 4 cores it has almost the same performance as the vanilla TM configurations with aggressive shutdown.

Aborts do not seem to be much of an issue with the genome benchmark, although it is suprising that the vanilla configuration performs so well despite a 34% overflow rate. Simulations limited to the first phase of the benchmark show that nearly all transactions overflow, seemingly indicating that all overflows in the full benchmark come from the first phase.

The most likely explanation for the worse performance by the VC configuration is the bus traffic: the bus is nearly saturated for the VC configuration with only 4 cores. Waiting for the bus requires more energy and slows down

A simulation suite with varying cache sizes showed that doubling the cache size halves the overflow rate, making the vanilla configuration more efficient. Meanwhile varying the cache size does not affect the VC configuration to any noticeable extent.

5 Future benchmarks

It may be useful to implement another benchmark, although none have all the traits we want. We would like a benchmark that allows us control and decouple overflows and aborts while still using the transactional caches. Possibilities include *intruder*, which has a medium sized read/write set and short transactions, but high contention, and *yada*, which has large and long transactions with medium contention.

References

[1] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. Jun 2007.