QQ: Nanoscale Timing and Profiling

James Frye[+*] James G. King[+*] Christine J Wilson[◇*]
Frederick C. Harris, Jr.[+*]

[+]Department of Computer Science and Engineering
[*]Brain Computation Lab
[◇]Biomedical Engineering

University of Nevada, Reno NV 89557

Abstract

QQ is set of tools for timing and memory profiling capable of analysis at nanoscale time resolution. It is platform independent and useful for either sequential, distributed, or parallel programs. The output can be visualized using various graphics packages. Currently QQ utilizes the IA32 hardware architecture to monitor performance but can be readily ported to other hardware. QQ can be implemented and customized quickly and is scalable.
Keywords: profiling, nanoscale resolution, memory use

1 Introduction

When developing a complex program that requires massive computational power, proper tracking of performance is crucial for development. Effective optimization requires some means quickly and accurately measuring the effects of code changes.

The NeoCortical Simulator (NCS) [9, 10, 11] is such a complex system. It is a large-scale biologically realistic simulator of cortical neurons. It has successfully simulated biological neural networks with 10^6 cells and 10^9 synaptic connections [6]. One aim of NCS is to run simulations at speeds approaching real-time. Since NCS utilizes the fastest available hardware for processing and internodal communication, the code was the only place where we could make significant performance improvements. Therefore, considerable effort has been devoted to optimizing the code for both speed and memory utilization [2].

The available performance monitors that were reviewed included TAU Portable Profiling Package [5], and SvPablo: A multi-Language Performance Analysis System[7]. Both of these packages had steep learning curves and were not easily integrated in NCS. TAU is compact and able to trace individual user-defined code blocks. It also utilizes hardware counters, but relies on pre-compiling code to embed the timing commands using a pre-compiler that is not compatible with gcc. SvPablo utilizes hardware counters like QQ, but the graphical interface and meta-meta-format output was more than what was necessary for our analysis.

QQ was developed as a set of tools that can accurately measure either elapsed time or number of machine instructions executed and memory usage with a minimum of overhead. The name QQ is not an acronym but was chosen as a name prefix simply because the letter Q is seldom used as the first letter of a variable or function name and would minimize potential conflicts with existing code. QQ was used extensively in optimization of NCS, and has been applied to a number of other programs.

The organization of this paper is as follows: Section 2 will address the design considerations and implementation. Section 3 covers the analysis routines and defines QQ's features. Section 4 covers examples of QQ's use in a sequential and parallel example programs. Section 5 gives some concluding remarks and defines the future work and direction of development of QQ.

2 QQ Design

2.1 Nanoscale Timing

Straightforward measurement of NCS elements is difficult. Different algorithms contribute varying numbers of elements in an input-dependent sequence. This makes NCS unlike other compute-intensive programs that mainly consist of a few loops iterating many times over the same code. Performance is not adequately measured by a relatively coarse-grained tool such as gprof [1]. The documented gprof resolution is approximately a hundredth of a millisecond or 10^{-5} seconds. And current system timing routines (Linux `gettimeofday`, `MPI_Wtime`, etc.) typically measure to an accuracy of, at best, microseconds (10^{-6} seconds). At current processor speeds, this translates to a granularity of several thousands of instructions.

NCS demanded precise timing measurements on the order of tens or hundreds of machine cycles, or 10^{-9} seconds. Therefore, without the resolution provide by the QQ tools, the effect of a change to any single element may have been overlooked.

2.2 Hardware Time Stamp Counter

Most modern microprocessors include a hardware based time stamp counter. NCS is currently compiled for use on the Intel IA32 (Pentium) architecture. Each processor contains a 64-bit hardware counter, which is zeroed when the processor is booted, and thereafter increments every clock cycle. At current processor speeds a 64-bit counter would roll over about once a century. The IA32 instruction set provides the `RDTSC` instruction, which stores the time stamp counter contents into the `edx:eax` register pair. Subtracting the values of two successive calls to this instruction thus gives a count of processor cycles. The actual elapsed time is derived by division by the clock frequency and is theoretically accurate to the processor clock frequency, or (at current CPU speeds) a fraction of a nanosecond.

Factors such as pipelining and out-of-order execution reduce this theoretical accuracy to several tens of clock cycles. Although steps such as issuing serializing instructions bracketing a code section will eliminate this factor. This serializing will alter the timing of the code being measured.

By itself, the `RDTSC` instruction, or its cognates in other architectures, provides a simple means of measuring the elapsed time between two points in a program. In itself, this is a useful and important performance measurement.

2.3 Sequential Profiling

Frequently, detailed information is desired, for example, marking the time at which a particular event occurs, or tracing the amount of time spent within a repeatedly called section of code. It is also desirable to somehow label or mark the different timed event. naturally, we would like to do this with a minimum of overhead both in programmer workload and execution time. Instrumentation calls were designed to be included in a non-obtrusive way and included in the code via a simple compile flag `QQ_ENABLE`.

QQ is based on the notion of tracing named events. Each event is characterized by a key, which identifies the particular event, and an event time. In addition, depending on the type of event, a value, count or state flag may be recorded. The different event types are combined into an event union.

When the profiler is initialized by calling the `QQInit` function, it allocates memory for a user-specified number of events, initializes the event pointer to the first event, and sets the base time to the current `RDTSC` value.

After initialization, one or more of the `QQAdd*` functions are called to add event types to the internal name table. Each function is passed a name (character string) identifying the event, and returns the integer key for the event. These keys are variables in the program space, and are the only remnants of QQ that remain in code when compiled with the `QQ_ENABLE` flag off.

The individual event recording functions are thus reduced to a minimum: each checks to see if the event pointer has overflowed the allocated buffer. If not, the `RDTSC` instruction is called, the key, count, and any other information is written to the current event, and the event counter is incremented.

The `QQRecord` function allows event recording to be turned off and on under program control. It does this by caching the current event pointer and replacing it with a value greater than the maximum allocated, thus allowing the event recording functions to use a single test to determine whether or not an event is to be recorded.

When profiling has finished, the `QQOut` function is called to write the saved event information to a file.

2.4 Parallel Profiling

For parallel programs, QQ takes some additional steps. First, `QQInit` will do an `MPI_Barrier` call to synchronize, as nearly as possible, timings on all nodes. Each node has its own event buffer, which records an independent set of events. On output, the buffers from all nodes are combined into a single file, along with information to identify which event belongs to which node.

2.5 Profiling Memory Use

2.5.1 Profiling Memory Allocation

Profiling memory usage is more difficult than profiling execution time. At this time a completely satisfactory solution seems impossible. However, QQ manages to gather much useful information. For example, in a Linux system, it is possible to obtain the total memory allocated to a process at any point in time by reading the `statm` pseudo-file in the process directory of the `/proc` filesystem. From this file, the number of 4 KByte pages allocated to the process can be read. While this number includes both code and data space, and may include the total memory allocated rather than what is currently in use, this value can be considered an upper boundary value.

The `GetMemoryUsed` function uses this method to return the total memory allocated to the program at the time when the function is called. By recording memory usage values before and after a block of code, one can obtain the number of pages being used by that block. However, it does not allow for direct measurements on the scale of structures or objects.

The system allocates memory to a program in units of pages. It is up to the internal memory allocator (generally in the `malloc` library) to parcel the pages into smaller units. If this memory allocation is done directly by calls to `malloc` library functions, it is easily measured by using the preprocessor to redefine the function call to a different function, which records allocation information and passes the operation through to the actual allocation function.

Thus, for example, the `malloc` function can be redefined to be:

```
#ifdef MEM_STATS
    #define malloc(arg) MemMalloc (MEM_KEY, arg)
#endif
```

Each call to malloc in the code now becomes a call to the `MemMalloc` function, and the new call contains an additional argument, which is the key under which the allocated memory will be recorded. Each chunk of allocated memory is stored in a C++ `map` object, indexed by its address (the pointer value returned by the malloc call). The call to `free` is redefined to remove the item corresponding to the address from the map. In this way, a consistent record of all currently-allocated memory is maintained, and the record for each item may contain information on the type of object, what routine allocated it, and so forth.

2.5.2 Monitoring Object Creation

In principle it should be possible to do something similar to the above for objects by overloading the new and delete operators, although we have not yet done so. Instead, a simpler method was used.

Each object constructor must have calls to the `MEMADDOBJECT` and `MEMFREEOBJECT` routines added to them. When memory profiling is turned off, these calls evaluate to empty statements; when it is on, they evaluate to calls to the memory recording functions and contain the object's this pointer and the `sizeof(this)` value as arguments.

These functions allow most explicitly allocated memory and objects defined in the code, to be recorded. Memory that is allocated internally by various library functions or standard C++ objects can not be recorded, however. The information available, although not complete, is still quite useful. If nothing else, the amount recorded as allocated by the profiler can be compared to the total memory allocated to the program. If the two quantities are significantly different, the memory is probably being allocated somewhere internally.

3 Analysis Routines

The timing interface for QQ includes many functions for tracing temporal events. Figure 1 shows the complete application programing interface currently available.

Since the QQ tools are intended for programmers, the output was formatted to be flexible. Most programmers prefer to write their own custom analysis code, or pipe the output to a general purpose tool such as gnuplot [8], for example. Unlike similar applications, the output was created

Prototype	Description
`void QQInit (int nEvents);`	Initialize the package, specifying the max number of events to be recorded.
`void QQBaseTime (void);`	Reset the base RDTSC value in QQInit. This might be done if for instance there was a substantial amount of time in program initialization before the section of interest begins.
`int QQAddMark (char *);`	Add a mark type event to the table. The simplest QQ event type. Marks the time the event occurred.
`int QQAddState (char *);`	Add a state type event to the table. Keep track of a state, either on or off. Useful for monitoring when a code block is entered (state on), then exited (state off).
`int QQAddCount (char *);`	Add a count type event to the table. Records the event time along with an associated integer.
`int QQAddValue (char *);`	Add a value type event to the table. Records the event time along with an associated floating point value.
`void QQMark (int key);`	For use by Mark type events, records the current time.
`void QQStateOn (int key);`	For use by State type events, indicates that the event has begun along with the current time.
`void QQStateOff (int key);`	For use by State type events, indicate that the event has finished along with the current time.
`void QQCount (int key, int count);`	For use by Count type events, record the current time along with an integer relevant to the event.
`void QQValue (int key, double value);`	For use by Value type events, record the current time along with a floating point value relevant to the event.
`void QQRecord (int flag);`	Allows for the recording of events to be turned on/off (flag=TRUE or FALSE) during program execution. Recording is on by default.
`void QQOut ( char *filename, int, int );`	Event information should be output to the specified filename, indicating this node's rank among the node count.

Figure 1: QQ API

to be flexible, precise and detailed, rather than processed via predefined functions. To this end, the output is stored in binary in the following format.

Each event to be traced is given a key. In the output file, the number of keys `nkeys(int)` is followed by the length of the key name string, `keylen (int)`. Then for each key, the following information is output, the index of the key, the key type and its name. Then, the information regarding the nodes is output. The number of nodes `nnodes(int)` is defined. For each node the following information is output, the offset or index in file at which the data for the node starts, the number of entries for the node, the base tick count for the node and the frequency of the node in MHz. Once this information has been defined, the data for each node is output.

During the evaluation of NCS the following two programs were created and are included as examples:

- Summary statistics is a simple C program which reads the QQ output file and produces a summary report of the time spent in each state, the number of times each event or state, etc.

- Profile viewer is a simple graphical application. Data is read from the output file, and piped to `gnuplot` for display. A simple interface allows the user to select which nodes, events, and time ranges are displayed.

4 Examples

Now that we have given an overview of QQ, some examples will help show how it can be used to profile and optimize code. In this section we present two examples, a sequential piece of code and a large parallel piece of code. The first will show how the code must be modified to be used and the second will give a big picture overview of what we were able to accomplish with QQ.

4.1 Sequential Code: BCS

BCS, the Brain Communication Server, is a companion program intended to coordinate data flow between NCS and a separate client. These clients under development make use of NCS in some way such as a virtual organism in a virtual environment, or eventually robotic actuators maneuvering in the real world.

Ideally, this server should have minimal impact on the simulation's execution while transferring the necessary information securely and accurately, so it also needs profiling for optimum performance. Even though BCS is a sequential program, it can still use QQ to evaluate code sections even without QQ's parallel features.

Inserting QQ into the program is easily done. Certain profiling needs to be performed in the main file of the program as it started. The code segment in Figure 2 shows the few lines needed to perform the profiling.

```
QQInit(1000); //create space for large number of possible events
QQBaseTime(0); //reset timer

//create new event to monitor load functions
QQload = QQAddState( "load" );

QQStateOn (QQload);  //Begin monitoring

loadAppList();

loadScriptList();

QQStateOff (QQload); //End monitoring
```

Figure 2: Basic QQ profiling

BCS makes use of an Object Oriented design, and the example code given in Figure 3 show how a QQ event can be declared once in the main routine, but used in a class defined elsewhere. By putting the event declaration in the main file, we ensure that multiple class instances don't try to add the same event multiple times and we acquire a single event key to be shared by all instances of the class. This shared event key could also be created using a static class member variable, but this example is not shown in order to be concise.

When the profiling is complete, the QQ output file is generated with a single function call. This is illustrated in Figure 4.

The simple and consistent layout of the QQ output file allows for the collected data to be extracted easily. Once the time values are read, they can be manipulated in whatever way the programmer prefers. Using all data, or selecting only a portion to view, it is easy to handle and forward to an appropriate process. In Figure 5 they are put into a table by the previously mentioned Summary statistics program. In the case of BCS, some of the events take up more time than the similar "gather" event. The events in question can be further broken down to better profile within the code blocks and pinpoint where inefficient code exists.

```
// from main file:

int gatherTime;  //global scope
...
gatherTime = QQAddState( ‘‘gather’’ );  //inside a main function
```

```
// from class file:

extern int gatherTime; //declared in main file

class function() \{
   QQStateOn( gatherTime ); //turn on state at start of block
   ...
   QQStateOff( gatherTime ); //turn off state at end of block
}
```

Figure 3: QQ Events in classes: Create and Use separately

```
// when monitoring is finished:
QQOut( "myProfile", 0, 1 );
```

Figure 4: Code for QQ Output illustrated

When performing the profiling, the program should be compiled with the value `QQ_ENABLE` defined (in gcc this is done by adding `-DQQ_ENABLE` to the command line). Later, by removing `QQ_ENABLE`, the profiling functions can stay in the source code, but will perform no operations. Even though a large number of events may have been originally declared for initialization of QQ, the events will no longer take up that amount of memory. In fact, the only remnants are the integers declared by the user to store the event keys.

```
File myProfile:   1 nodes

Node   0:  2592.403 MHz, 10 keys, 122 states, ticks from 15956 to 105435813502

           ET = 40.671068 sec

State outputs: Counts are millions of cycles

      Hits        Time    Percent      Name
 1:      2     0.000889   0.002%  'load'
 2:     46     0.562092   1.382%  'setpath'
 3:      8     0.159862   0.393%  'getdata'
 4:      8     0.166010   0.408%  'gettime'
 5:      2     0.000241   0.001%  'setpattern'
 6:     20     0.953694   2.345%  'launch'
 7:      8     0.158786   0.390%  'reportcount'
 8:     20     0.385492   0.948%  'mkdir'
 9:      8     0.000237   0.001%  'gather'
```

Figure 5: QQ output displayed by st

4.2 Parallel Code: NCS

For our parallel example we will discuss QQ's usage with NCS, the application that actually led to the design and implementation of QQ [2]. NCS is a neocortical-neural network simulator which incorporates laboratory-determined synaptic and membrane parameters into a large-scale, biologically realistic model of cortical modules. Results to date have demonstrated biological accuracy in synaptic and membrane dynamics, and suggested that computational models of this scope can produce realistic spike encoding of human speech [3, 4]. Efficient parallel processing of this very large-scale program is crucial to a realistic model which executes in a practical time frame.

NCS has gone through a number of development cycles. Here we discuss optimizations done between NCS3, the first parallel, cluster-based code, and NCS5, the current working version.

4.2.1 Optimization Targets

Four factors affect the performance of NCS: load imbalance, message-passing overhead, and synchronization on the parallel side, and the sequential performance of the code executing on each compute node. All of these areas were addressed in the optimization process. This section describes that process and the solutions that were developed.

Load Balancing For many applications load balancing is a simple matter of assigning an equal number of work units to each node. Version 3 of NCS (NCS3) used this method, but it proved less than satisfactory in practice, due to the rather unusual characteristics of the process being modeled.

Unlike many parallel models which operate over a grid of similar entities, NCS contains algorithms which model many different neural components. These components may be combined in different ways to create many different types of cells, and those cells may be connected in fairly arbitrary ways. There are thus few if any points in the code where we can measure the repeated execution of a single component type.

Much of the computation time is devoted to modeling individual synapses. Factoring these into the load-balancing process is complicated by the fact that the computation takes place on a particular synapse only when the synapse is in a firing state. Only a small fraction of synapses will be in this state at any particular timestep, and it is not possible to predict which synapses will fire, because firing is determined by the input stimuli.

This unpredictability applies to memory usage as well: the amount of memory needed to construct a brain is the sum of its components, but a running brain needs significant additional memory to hold dynamic information for synapse firing states. The exact amount required is impossible to predict, but in practice the current implementation seems to require about equal amounts of memory for static and dynamic data.

The total amount of computation is determined by assigning some weight, say 1.0 for simplicity, to a basic cell and ratios of this amount to other components that may be added. Using QQ we were able to time individual components and create a weight/cost table for them. Thus a simple summation gives a total compute weight for each cluster of cells. These clusters can then be assigned to nodes according to some scheme which balances the compute load according to the computing power available on each processor.

In order to run the largest models it is necessary to balance memory use, rather than compute load, and accept the resulting inefficiency. Distribution follows a similar algorithm, merely substituting the memory footprint of each component for its compute weight.

Message-Passing Overhead Initial profiling of NCS3 determined that much time and memory were devoted to message passing. More detailed examination disclosed a number of inefficiencies in the message passing scheme. The most notable was the pre-allocation of messages, with a 60-byte message object allocated for every synapse. On a large model of a billion synapses or so, this consumes nearly a quarter of the 256 GBytes of memory on our current cluster. Since only a small fraction of synapses (typically less than 1%) are actively firing (and thus transmitting a message) during any particular timestep, most of this memory was actually unused.

Other inefficiencies related to the use of the same communicator and message format for distributing stimulus and report data and the synapse firing messages. This required the inclusion of a message type field in the message packet, as well as additional overhead needed to distribute messages of different kinds to the proper destinations.

NCS5 separates the three functions. Stimulus messages and reports (excepting real-time I/O) are now produced locally on each node, which reduces the traffic on the network and, along with other optimizations, allows the size of the individual synapse firing message to be reduced from 60 to 20 bytes. In the NCS3 message packaging scheme, each message transmitted about 40 bytes of unnecessary information, resulting in a 200% overhead.

While these changes improved performance significantly, further analysis showed that yet more improvement was possible. The old algorithm passed message objects through several layers, so that a typical message was being read and written perhaps five times or more in its progress from source to destination.

In the optimized scheme, the message has no existence as an individual object. It is instead a logical entity within a packet containing many individual messages. so that the bulk of the information in a message can be written once, when sent, and read once, when it is received at its destination. Instead of individual messages, the program deals with packets containing many messages. Not only does this eliminate much overhead, it means that the packet size can be chosen to match the most efficient transfer size of the underlying hardware.

Synchronization Most of the computation in NCS is devoted to determining the effects of synapse firings on the receiving compartments. These firings are essentially unpredictable, being determined by the brain's reactions to stimulus propagating through a highly nonlinear feedback network. Therefore it can be expected that, regardless of how well the number of synapses is balanced between nodes, the actual amount of computation will vary both between nodes and over time.

As a consequence, one node, and probably not the same node at each timestep, will take the longest amount of time to finish its computations. If a simple end of timestep barrier is used for synchronization, then all the other nodes will be idle for some part of the timestep.Figure 6 shows an example of this idle time, indicated by the horizontal lines at Y=1. Node 1 has (for the displayed timesteps) the heaviest load, and so displays little or no idle time (labeled `MessageBus::Sync` in the figure), while the others display more, with the amount varying between nodes and between timesteps.

We used this data to redesign the MessageBus. Recall from basic biology that the electrochemical pulse from a firing cell propagates along its synapses at a relatively slow speed, so that the

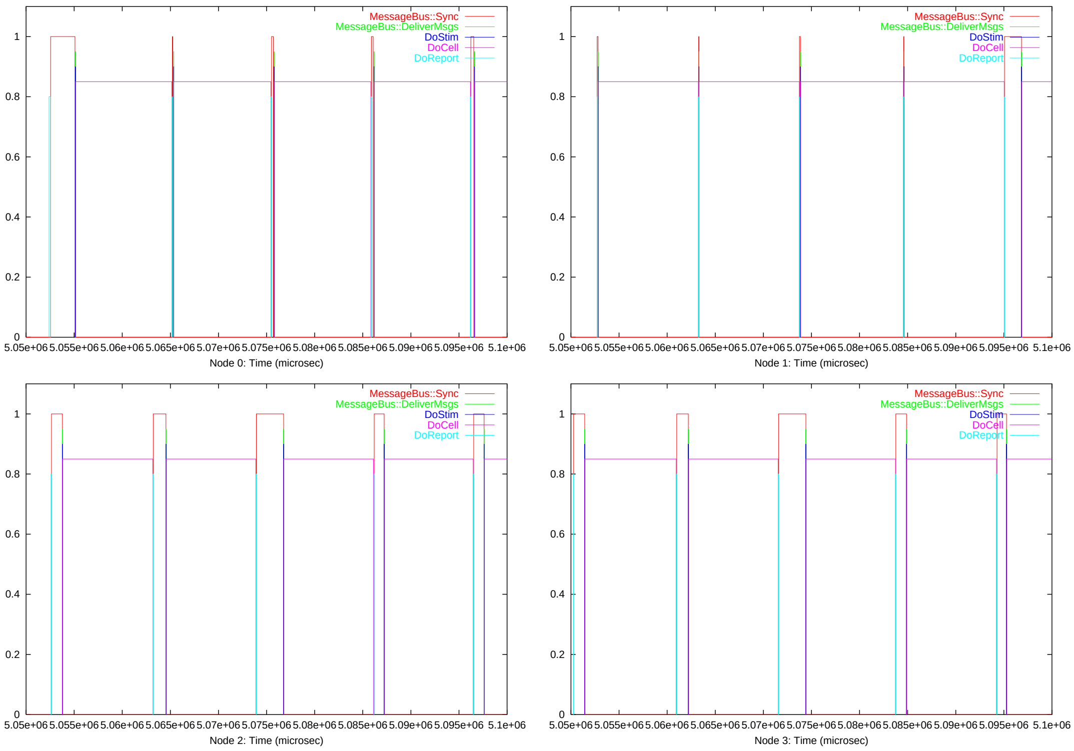

Figure 6: Idle Time Due to Load Imbalance.

transmission time between the sending and receiving cells typically translates to several tens of simulation timesteps. Thus for each node there is an event horizon, which depends on the minimum synapse propagation time of the nodes with which it communicates. If this minimum time is dt, then nothing other nodes do at time T can affect this node until time $T+dt$. Therefore, a barrier mechanism constructed to utilize this event horizon can allow some of the end-of-timestep idle time to be used. A node may simply continue to work until it reaches $T+dt$. Meanwhile, messages have continued to arrive from the other nodes, and unless the node is consistently under-loaded, these messages will contain `SYNC` flags indicating that their nodes have progressed to another timestep.

Synchronization now becomes a relatively simple matter. On initialization, a `NodeTime` array is allocated, with entries for each node from which the node receives messages. As `SYNC` packets are received, these times are updated. When the node reaches the end of each timestep, these NodeTimes fields are checked. If the other nodes are within the minimum time difference, then the node can proceed to the next timestep; if not, it must wait for more packets to be received and check again.

4.2.2 Results

In addition to the algorithmic improvements described in the preceding subsection, many other optimizations have been made in the process of creating version 5 of the NCS program. These are too numerous to describe individually. Indeed, most are obvious, and of no great theoretic interest. This section describes the cumulative effect of all optimizations.

It is difficult if not impossible to define a simple performance metric for NCS. For a number of reasons, the time a particular NCS brain takes to process some input file is only a useful performance measure for that particular brain design and input.

One reason is that NCS defines many different components, which the user may include in fairly arbitrary proportions and connect in a large number of ways. Since the behavior of NCS is highly non-linear, these differences can result in large variations in processing time for models which might appear superficially similar.

The approach we take here is to measure the performance of particular functional areas, or groups of operations with similar characteristics. Because the groups share common performance features, the effect of a change in the area on the whole program can be estimated. The area's speed change can be compared between program revisions. We then relate the changes to total run time on the same input.

We have tested with many models but due to space we will only present some of the results from the simple model `1Column` model. This model has a single column of three layers, each having two cell types, `excitatory` and `inhibitory`. Input is from an artificial pulse stimulus, which causes it to exhibits an unrealistically high cell firing rate. Figure 7 shows the time usage of the components in a one simulated second run of the 1Column model just described. Note that we have expanded the figure portion for NCS5 since the optimization was quite successful.

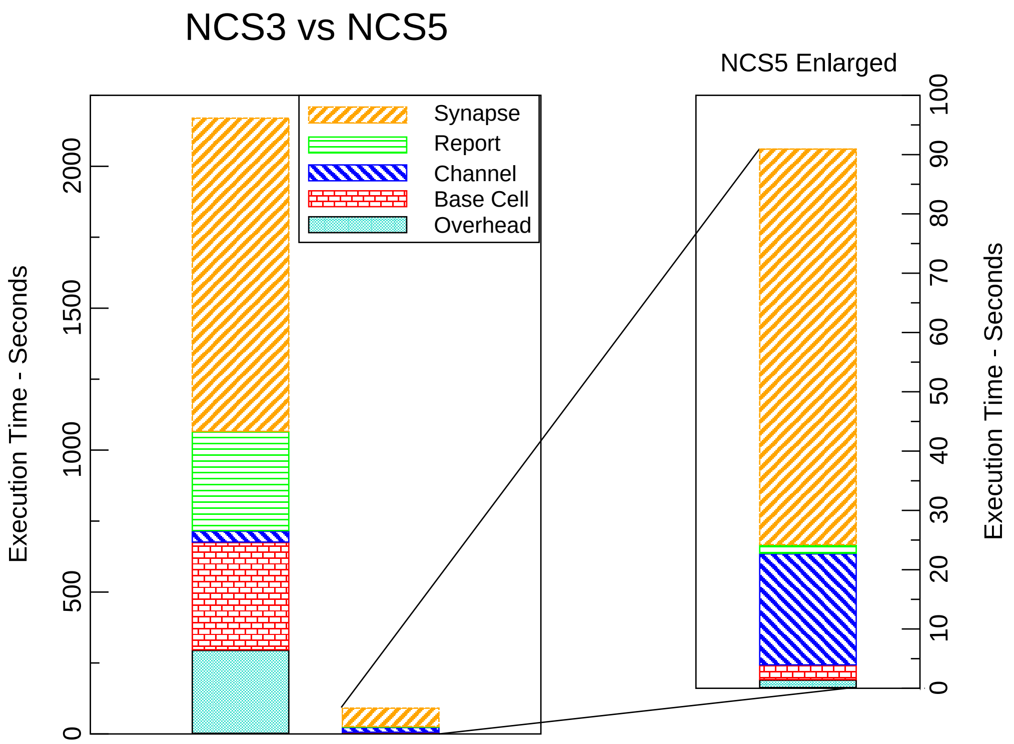

Figure 7: Share of CPU Time Used by Functional Areas, 1Column Model.

5 Conclusions and Future Work

In conclusion, we have developed and tested an new profiling tool that allows nanoscale timing of code segments, profiling, and memory usage analysis. Using this tool, we have decreased NCS run time by nearly two orders of magnitude, and improved memory utilization several-fold. This optimization is not complete: we have observed remaining memory access patterns and code bottlenecks whose improvement we expect to yield a further order of magnitude improvement.

Although developed specifically to optimize NCS, QQ has proven to be of value in the profiling and optimization of a number of other programs. The temporal resolution allows for fine-grained measurements of specific events or blocks of code. It can be used on sequential and parallel programs without modification.

Accessing the hardware time stamp counters of other architectures would extend the usability of these tools. Also, the ability to measure memory at an object or event level with a small memory and performance footprint, is an area that requires additional work. Currently however, we have added only those features we actually use and can implement easily since our focus is on NCS development. We believe this has kept QQ both simple and effective.

References

[1] Free Software Foundation. gprof user manual. `http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html` (10/27/04).

[2] James Frye. Parallel optimization of a neocortical simulation program. Master's thesis, University of Nevada, Reno, December 2003.

[3] J. C. Macera, P. H. Goodman, F. C. Harris, Jr., R. Drewes, and J. Maciokas. Remote-neocortex control of robotic search and threat identification. *Robotics and Autonomous Systems*, 46(2):97–110, February 2004.

[4] James B. Maciokas. *Towards an Understanding of the Synergistic Properties of Cortical Processing: A Neuronal Computational Modeling Approach*. PhD thesis, University of Nevada, Reno, August 2003.

[5] University of Oregon. Tau portable profiling. `http://www.cs.uoregon.edu/research/paracomp/tau/tautools`.

[6] M.C. Ripplinger, C.J. Wilson, J.G. King, J. Frye, F. C. Harris, Jr., and P.H. Goodman. Computational model of interacting brain networks. American Federation of Medical Research Conference, January 2004.

[7] Luiz De Rose, Ying Zhang, and Daniel A. Reed. SvPablo: A multi-language performance analysis system. *Lecture Notes in Computer Science*, 1469, 1998.

[8] Thomas Williams and Colin Kelley. gnuplot user manual. `http://www.gnuplot.info/docs/gnuplot.html` (11/29/03).

[9] E. Courtenay Wilson. Parallel implementation of a large scale biologically realistic neocortical neural network simulator. Master's thesis, University of Nevada, Reno, August 2001.

[10] E. Courtenay Wilson, Phillip H. Goodman, and Jr. Frederick C. Harris. Implementation of a biologically realistic parallel neocortical-neural network simulator. Proc. of the 10^{th} SIAM Conf. on Parallel Process. for Sci. Comput., March 2001.

[11] E. Courtenay Wilson, Frederick C. Harris, Jr., and Phillip H. Goodman. A large-scale biologically realistic cortical simulator. Proc. of SC 2001, November 2001.