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
{frye, wilsonc,king,fredh}@cs.unr.edu

Abstract

QQ is a tool for timing and memory profiling capable of nanoscale time resolution. Designed to minimize both learning curve and impact on the profiled code, it is platform independent and usable with sequential, distributed, and parallel programs. QQ is invoked via embedded function calls, and outputs event and timing records in a compact binary format. These records can be analyzed by external software packages. QQ obtains timing information from the hardware performance monitoring facilities designed into current microprocessors. Our implementation addresses the IA32 architecture, but the similar facilities of most modern processors allow QQ to readily be ported to other platforms.

Keywords: *profiling, nanoscale resolution, memory use*

1. Introduction

The NeoCortical Simulator (NCS) [7, 8, 9] is a large-scale biologically realistic simulator of cortical neurons. On a 128 processor Beowulf cluster with 256 GBytes of RAM, it has successfully simulated biological neural networks with up to 10^6 cells and 10^9 synaptic connections [5]. One goal of our research program has been to run such simulations at speeds approaching real-time, but they now require many minutes or hours of cluster time for each simulated second. Since Moore's Law (processor speeds double every 18 months) predicts a 15 year wait for the necessary hardware performance, we have devoted considerable effort to optimizing the code for both speed and memory utilization [1].

An NCS simulation may contain many millions of individual objects (of several dozen basic types), each modelling some component of a brain cell, and all connected in a highly nonlinear feedback network. Input stimuli propagating through the network cause the individual objects to react. At any point in time, only a fraction of objects are computationally active, but which of them, and what form their computation takes, varies unpredictably from timestep to timestep.

Effective optimization requires the ability to accurately measure changes in the performance of the code being optimized. Given the design of NCS, this requires measuring changes in the performance of single executions of individual objects. We examined a number of performance monitoring packages. Few of them were able to achieve the necessary resolution; those that could would have required major changes in our development environment.

A similar situation existed with respect to memory: we found a number of packages that would monitor program memory use and help locate conflicts and memory leaks, but none that would resolve allocations at the scale of individual objects.

We therefore developed QQ: a set of tools that can measure performance to the level of single machine instructions, and memory use to individual allocations. QQ is used extensively in ongoing NCS optimization, and has been applied to a number of other programs.

This paper is organized as follows: Section 2 addresses design considerations, related work, and implementation. Section 3 covers the analysis routines and defines QQ's features. Section 4 shows examples of QQ's use in sequential and parallel programs. Section 5 gives some concluding remarks and defines the future work and direction of development of QQ.

1.1. Related Work

Before developing QQ, we examined a number of performance monitoring packages, but none came close to meeting our needs. Most, like the standard `gprof`, were simply

not capable of the required accuracy, or added unacceptable run-time overhead, and so were rejected after superficial examination.

Two packages deserved further examination: TAU [4] and SvPablo [6]. Both appeared capable of the required accuracy, but both were ultimately rejected due to our inability to integrate them into the NCS development environment.

TAU is compact and able to trace individual user-defined code blocks. However, it uses a pre-compiling step to embed timing commands in the source, and this pre-compiler is incompatible with the version of C++ used by the gcc compiler.

Like QQ, SvPablo utilizes hardware counters for maximum accuracy, but the graphical interface and meta-meta-format output imposed what proved to be an unacceptably steep learning curve.

2. QQ Design

Modern microprocessor architectures implement a variety of on-chip debugging and performance monitoring features, and it is here that we found the key to nanoscale timing resolution.

2.1. Hardware Time Stamp Counter

NCS development is presently done on an Intel IA32 architecture (Pentium) platform. IA32 performance monitoring support includes a 64-bit processor *time stamp counter* which is zeroed when the processor is booted, and increments every clock cycle. User code can read this counter via the IA32 assembly language `RDTSC` instruction. Subtracting two successive reads gives a count of processor clock cycles, and division by the processor clock frequency gives an elapsed time that at current CPU speeds is accurate to a fraction of a nanosecond.

Actual code timings are somewhat less accurate than this, due to hardware optimization. The processor decodes each instruction into μops, which are re-ordered and executed in parallel by one of several execution units in a manner that depends on the instruction context. In practice we see granularity on the order of a few tens of clock cycles, which is some two orders of magnitude better than other timing methods.

The `RDTSC` instruction (and its cognates in other architectures) thus provides a simple means of measuring the elapsed time between two points in a program. This in itself is a useful performance measurement.

2.2. Sequential Profiling

A useful profiling tool requires more than a simple timing routine. It should be able to time many sections of code, accumulate the total time spent within repeatedly called sections, mark the time at which events occur, and so on. QQ does this by recording named events. Several types of events are defined. Each consists of an integer key that identifies the event, and an event time, which is simply the value returned by the `RDTSC` instruction. Depending on the type, a value, count or state flag may also be recorded.

It should also require a minimum of overhead in both execution time and programmer workload. QQ's instrumentation calls are therefore designed to be as non-obtrusive as possible. They are included in code as simple function calls. A single compile flag, `QQ_ENABLE`, determines whether profiling is enabled. If it is not, the definitions in the QQ.h include file reduce all the calls to no-ops. Thus when profiling is turned off, there is no effect on execution speed, and only a very small additional memory footprint.

QQ is initialized by calling the `QQInit` function (see Section 3). This 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 specify events to be recorded. Each function takes an identifying name, and returns a corresponding integer key. These keys are variables in the program space, and are the only trace of QQ that remains in code compiled with the `QQ_ENABLE` flag off.

In order to minimize the impact of profiling on program timing, event recording functions are reduced to a minimum. Each checks to see if the allocated event buffer has overflowed. If not, the key and `RDTSC` return value are written to the buffer (along with any other information defined for the type of event), and the event pointer is incremented.

During program execution, event recording may be turned off and on by the `QQRecord` function, so that only area of interest need be profiled.

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

2.3. Parallel Profiling

For parallel profiling, QQ takes some additional steps. First, `QQInit` does 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.

2.4. Profiling Memory Allocation

Profiling memory use is more difficult than profiling execution time. At this time a completely satisfactory solution

seems impossible, since allocation is often hidden within library calls and object constructors. However, QQ manages to gather much useful information.

2.4.1. Gross Memory Allocation On a Linux system, the number of 4 KByte memory pages currently allocated to a process can be read from its `statm` pseudo-file in the `/proc` filesystem. This number includes both code and data space, and may include memory allocated but not currently in use, so it must be considered an upper boundary value.

The `GetMemoryUsed` function uses this method to return the total memory allocated to the program. By comparing values before and after a block of code, one can obtain an idea of the memory allocated by that block. This is a gross measurement, most useful when the block is allocating many megabytes.

2.4.2. Fine Memory Allocation The operating system allocates memory to a program in units of pages. The internal memory allocator, generally the `malloc` library, then parcels this memory into smaller units as required. When the program allocates memory by calling `malloc` library functions directly, it is easy to record allocation information: a preprocessor macro redefines the library calls in the program, and the redefined function records information before passing the operation through to the actual allocation function.

For example, the `malloc` function can be redefined as:

```
#ifdef MEM_STATS
  #define malloc(arg) Malloc (KEY, arg)
#endif
```

Each malloc call in the code now becomes a call to the `Malloc` function. The new call contains an additional argument, a key under which the allocated memory will be recorded. Each allocation is recorded, indexed by its address. The call to `free` is similarly redefined to remove the item from the map. In this way, a consistent record of allocated memory is maintained, with the record for each item containing information on the type of object, what routine allocated it, and so forth.

2.4.3. Monitoring Object Creation In principle the above method could be extended to objects by overloading the new and delete operators. In practice this is not easily done, so we use a simpler method. We add calls to the `MEMADDOBJECT` and `MEMFREEOBJECT` routines to the constructor and destructor of each object we want to monitor. With profiling off, these calls evaluate to empty statements; when it is on, they become calls that pass the object's `this` pointer and the `sizeof(this)` value to the memory recording functions.

These methods allow monitoring of explicitly allocated memory and objects defined by the program. Although memory allocated internally by library functions or standard C++ objects is not recorded, such memory has not been a significant factor in NCS development.

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.

The QQ tools were designed by and for programmers who often prefer to write custom analysis code. The output format was therefore designed to be flexible, precise and detailed, but still easily accessed.

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

Prototype	Description
`void QQInit (int nEvents);`	Initialize package, specifying a maximum number of events to record.
`void QQBaseTime (void);`	Reset the base time to current time. This is useful when a significant amount of unprofiled code is executed after the QQInit call.
`int QQAddMark (char *);`	Define a MARK event, which associates the current time with `key`.
`int QQAddState (char *);`	Define a STATE event, which records the start (state = ON) and end (state = OFF) of state `key`.
`int QQAddCount (char *);`	Define a COUNT event, which associates the time and a caller-supplied integer value with `key`.
`int QQAddValue (char *);`	Define a VALUE event, which associates the time and a caller-supplied `double` value with `key`.
`void QQMark (int key);`	Record the MARK event corresponding to `key`.
`void QQStateOn (int key);`	Record the start of STATE `key`
`void QQStateOff (int key);`	Record the end of STATE `key`
`void QQCount (int key, int count);`	Record COUNT event `key` and its associated integer `count`.
`void QQValue (int key, double val);`	Record COUNT event `key` and its associated `val`.
`void QQRecord (int flag);`	Enable or disable event recording during the program, according to the value of `flag`.
`void QQOut (char *name, int, int);`	Output the recorded profiling info to file `name`.

Figure 1. QQ API

external client programs. Such clients make use of NCS in some way, as for instance the brain of a virtual organism within a virtual environment, or eventually robotic actuators maneuvering in the real world. Ideally, this server should transfer the necessary information securely and accurately, with minimal impact on the simulation's execution. Therefore we used QQ profiling to measure and optimize performance.

QQ is easily inserted into the program. Figure 2 shows the few additional lines needed to to do initialization, time a code segment, and write the profiling data to a file.

```
QQInit (1000);        // Initialize QQ

QQload = QQAddState ("load"); // define event

QQStateOn (QQload);   // Record start time

loadAppList ();       // Do work...
loadScriptList ();

QQStateOff (QQload);  // Record end time

QQOut ("profile.qq", 0, 1);  // Write to file
```

Figure 2. Basic QQ profiling

To activate profiling, the program is compiled with `QQ_ENABLE` defined (in gcc this is done by adding `-DQQ_ENABLE` to the command line). Production code can be compiled from the same source simply by undefining `QQ_ENABLE`. Calls to profiling functions remain in the source code, but perform no operations. The only traces of QQ that remain in the executable are the allocation of a small amount of memory for the event keys, and assignments of 0 to each key.

BCS makes use of an object oriented design. The code in Figure 3 shows how a QQ event can be declared once in the main routine, but used in a class defined elsewhere. Declaring the event as a global variable causes the execution time of all class instances to be recorded under a single key. If instead we wished to profile instances separately, we could call `QQAddState` in the object constructor, of course adding some code to create a unique name for every instance.

The simple and consistent layout of the QQ output file allows the collected data to be easily extracted, and analyzed or displayed.

Figure 4 shows an example in which the output from a BCS profiling run is displayed as a simple table. 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 inefficient code.

4.2. Parallel Code: NCS

Here we discuss the use of QQ with NCS, the application that led its design and implementation [1]. 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 suggest

```
// A QQ event defined as global in the main routine...

#include "QQ.h"

int gatherTime;  // global scope

int main (int argc, char *argv [])
{
    ...
  gatherTime = QQAddState ("gather");
    ...
}

// And used in a class defined in a separate file.

extern int gatherTime; // declared in main file

class A_Class ()
{
  QQStateOn (gatherTime);   // start timing
    ...                     // execute some code
  QQStateOff (gatherTime);  // stop timing
}
```

Figure 3. QQ event created and used in separate classes.

```
File profile.qq:   1 nodes

Node  0:  2592.403 MHz, 10 keys, 122 states
          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 4. QQ output displayed by st

that computational models of this scope can produce realistic spike encoding of human speech [2, 3]. Models which reproduce such realistic behaviors typically require the simulation of 10^4 to 10^6 neurons, and 10^7 to 10^9 synaptic connections. Executing these very large models within a practical time frame requires extremely efficient parallel processing.

Neurobiology: NCS models the behavior of neurons. Like all cells, neurons maintain a voltage difference, the *membrane potential*, between their interior and the external environment. Unlike most cells, neurons are excitable: stimuli can cause *voltage spikes* in which the membrane potential changes from its *resting potential* of about -70 mV to +20 mV or so within a fraction of a millisecond, then returns to rest over several milliseconds.

A typical neuron has a cell body, or *soma*, and a long *axon*, which branches to make contacts with other neurons at *synapses*. A typical cortical neuron may make several thousand such contacts. A voltage spike, or *action potential* originates at the soma and propagates along the axon (at a speed which depends on the cell type) until it reaches the synapses, where it causes the release of *neurotransmitters*. These alter the membrane potential of the receiving cell, and, if the sum of the effects of all incoming synapses is sufficient, trigger a voltage spike in the receiving cell, which in turn propagates to other cells.

A collection of neurons thus forms an electro-chemical signaling network. NCS attempts to simulate the behavior of such networks.

4.2.1. Optimization Targets Four factors affect the performance of NCS: message-passing overhead, load imbalance, synchronization of parallel code, 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.

Message-Passing Overhead Recall from the discussion of neurobiology that NCS is simulating the propagation of action potentials between neurons. The behavior of these potentials is quite stereotyped: once initiated, a potential propagates along the axon at a constant speed and amplitude. It thus can be simulated by simply passing a message from sending to receiving cell. This message-passing mechanism, the *message bus*, in fact forms the core of NCS, and is responsible for its parallelism.

Profiling of early NCS versions determined that the message bus was using a large share of both memory and computation time. More detailed examination disclosed a number of inefficiencies in the initial implementation. The most notable was the pre-allocation of messages, with a 60-byte message object being allocated for every synapse. On a large (10^9 synapses), model this consumed nearly a quarter of the 256 GBytes of memory on our 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 at any given time.

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.

An improved message bus separated these 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. More importantly for memory consumption, message packets are allocated dynamically from a shared pool. This reduces memory use by an order of magnitude or more.

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. The bulk of the information in a packet thus needs to be written only once, when sent, and read once, when it is received at its destination. 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.

Load Balancing Load balancing is in principle a simple matter of assigning an equal number of work units to each node. For many applications, for instance those which operate over a spatial grid, it can be simple in practice. However, this is not the case for NCS. 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.

Even measuring any particular component's contribution to the compute load is problematic. There are few if any points in the code where we can measure the repeated execution of a single component type, so that an accurate measurement must resolve differences in a single call. Removing a component to measure the performance contribution is not possible, since such removal will generally cause a significant change the behavior of the system.

Furthermore, much of the computation time is devoted to modeling individual synapses. Factoring these into the load-balancing process is complicated by the fact that computation takes place on a particular synapse only when the synapse is in a firing state. Only a small fraction of synapses are 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, although in practice the current implementation seems to require about equal amounts of memory for static and dynamic data.

By using the nanosecond timing resolution provided by QQ, we can accurately measure the time required for a single execution of any component. These component times are entered in a weight/cost table, and adding the weights of all components give 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.

Synchronization Most of the computation in NCS is devoted to calculating the effects of synapse firings on the receiving compartments. These firings are essentially unpredictable, being determined by the brain's reactions to stimuli 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 to force all nodes to proceed in lockstep, then all the other nodes will be idle for some part of the timestep. Figure 5 shows an example of this idle time. For the displayed timesteps, Node 1 has the heaviest load and so shows little or no idle time (labeled `Idle` in the figure), while the others display more, with the amount varying between nodes and between timesteps.

We used this data to redesign the message bus. The biological action potential of a firing cell propagates along its axon at a relatively slow speed, so that the transmission time between sending and receiving cells typically translates to several tens of simulation timesteps. Since propagation is simulated by message passing, a firing cell can dispatch its outgoing message immediately. The message remains either in the MPI subsystem, or in a buffer on the receiving node, until its specified delivery time.

Thus for each node there is an event horizon, which depends on the minimum message 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

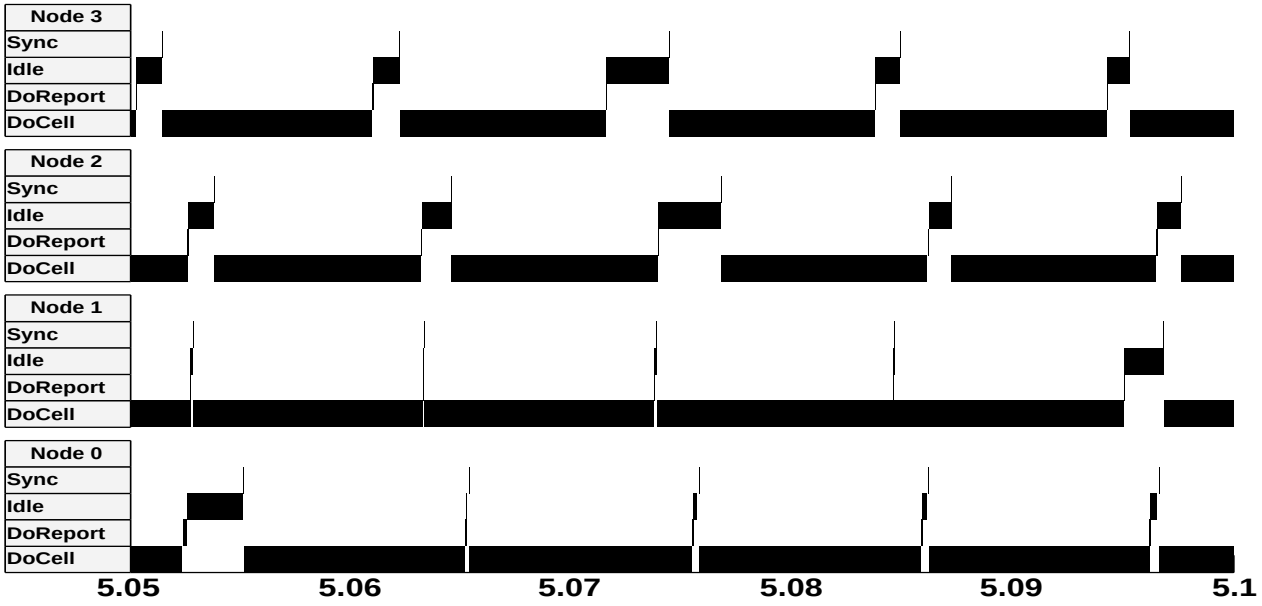

Figure 5. Idle Time Due to Load Imbalance.

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.

Sequential Optimization In addition to the algorithmic improvements described above, detailed profiling led to many sequential code optimizations. These are too numerous to describe individually.

4.2.2. Results It is difficult if not impossible to define a simple performance metric for NCS. 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. In large part this is because 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 the resulting network is highly non-linear, small changes in design can produce large variations in processing time, even 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`. 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 exhibit an unrealistically high cell firing rate. Figure 6 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.

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

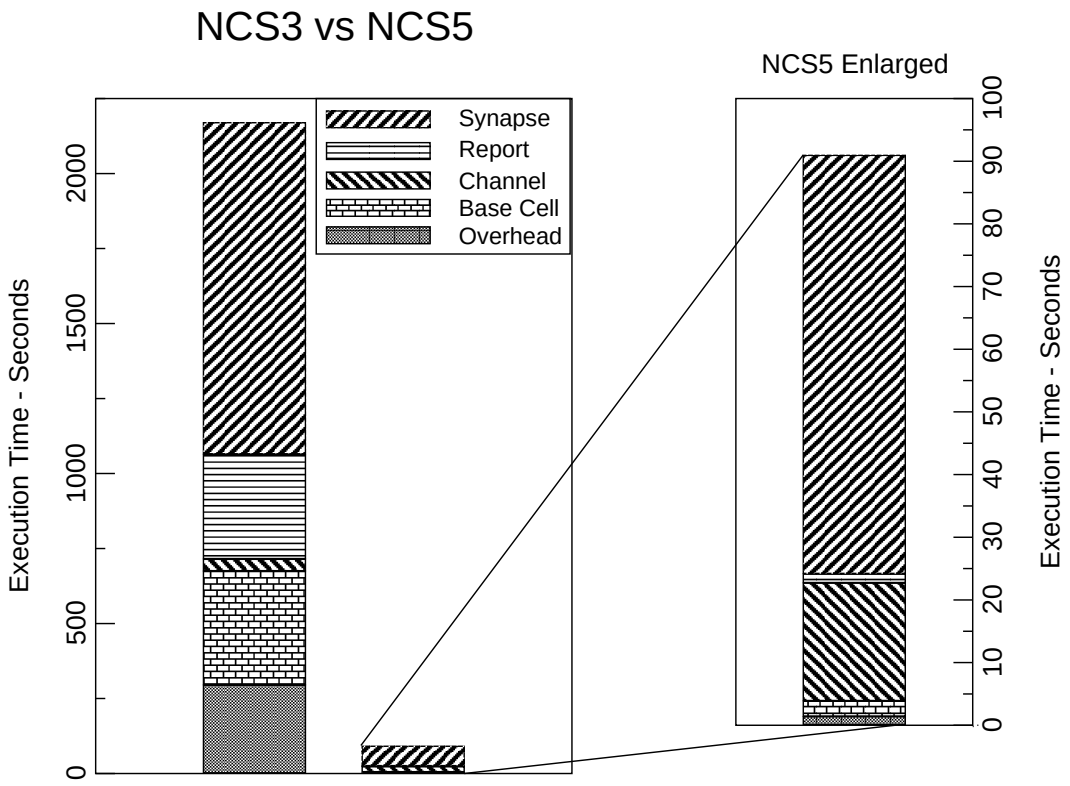

Figure 6. Share of CPU Time Used by Functional Areas, 1Column Model.

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 deserves additional work. Since our focus is on NCS development, however, we have implemented only those features that we actually intend to use. We believe this narrow focus has kept QQ both simple and effective.

QQ source, documentation, and examples may be found at `brain.cs.unr.edu/QQ`.

References

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

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

[3] J. 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.

[4] U. of Oregon. Tau portable profi ling. `http://www.cs.uoregon.edu/research/paracomp/tau/tautools`.

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

[6] L. D. Rose, Y. Zhang, and D. A. Reed. SvPablo: A multi-language performance analysis system. *Lecture Notes in Computer Science*, 1469, 1998.

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

[8] E. C. Wilson, P. H. Goodman, and F. C. Harris, Jr. 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.

[9] E. C. Wilson, F. C. Harris, Jr., and P. H. Goodman. A large-scale biologically realistic cortical simulator. Proc. of SC 2001, November 2001.