University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

C64prof: A Parallel Profiling Environment for the Cyclops64 Architecture

Mark Pellegrini
Guang Gao
{*mpellegr, ggao*}*@capsl.udel.edu*

CAPSL Technical Memo 106
June 28, 2011

Copyright © CAPSL at the University of Delaware

The following is the first version of this memo. Please ask for the author's permission to distribu te.

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Contents

1 Introduction **1**

2 Background **1**
2.1 Profiling . . . 1
2.2 The C64 architecture . . . 2

3 C64prof **4**
3.1 The C64prof API . . . 4
3.1.1 _noprofile_cygprofile_init . . . 5
3.1.2 _noprofile_cygprofile_shutdown . . . 6
3.1.3 _noprofile_cyclops_profile_begin_highpf . . . 6
3.1.4 _noprofile_cyclops_profile_end_highpf . . . 6
3.1.5 _noprofile_set_logfile_size . . . 7
3.1.6 _noprofile_set_max_recursion_depth . . . 7
3.1.7 _noprofile_get_max_recursion_depth . . . 7
3.2 PSHMEM - A Profiling Library for SHMEM . . . 8
3.3 Example program and C64prof trace . . . 8
3.4 Interpreting the C64prof trace . . . 11

4 Runtime Overhead **12**
4.1 Splash2 benchmark suite . . . 13
4.2 Finite-difference time-domain method . . . 13
4.3 Assembly-coded matrix-matrix multiply . . . 14

5 Ease of Use **14**

6 C64prof Data Visualization **15**
6.1 Single-pass mode . . . 15
6.2 Multi-pass mode . . . 17

7 Conclusions **17**

8 Future Work **18**

List of Figures

1 A logical view of the Cyclops64 architecture . . . 3
2 A diagrammatic view of the C64prof trace . . . 12
3 The C64prof visualizer in single-event mode synchronization graph for example.c . . . 15
4 The C64prof visualizer event count graph generated from example.c trace data (cropped view) . . . 16
5 The C64prof visualizer multi-event graph for example.c (cropped view) . . . 18

Abstract

In this paper, we describe C64prof, a profiler and visualizer we have created for the Cyclops64 architecture.

Motivated by a desire for an easy-to-use, low-overhead parallel performance analysis tool, we made use of two features of Cyclops64's environment - the hardware event counting library and the Cyclops64 compiler's automated instrumentation capability - to provide the programmer with a powerful, flexible tool for gathering program performance data. This data can be fed into the C64prof visualizer, which in turn can analyze the profile traces to produce digested output (including graphs).

We present three case studies using the profiler - two stand-alone applications (FDTD and Matrix-matrix multiply) and the SPLASH2 benchmarks to show that the profiler has low-overhead. Using lines of code added or modified as our metric, we also show that the profiler is extremely easy to use, typically requiring only a very small number of changes to code or makefiles.

1 Introduction

Cyclops64 is a high-performance multi-core architecture developed by IBM. Past experiments have shown that programs running on the Cyclops64 architecture can achieve very high performance, but experience has also shown that program execution and performance tends to be opaque and difficult to analyze. C64prof is a profiler designed to address this problem by giving the programmer a flexible, easy-to-use tool to allow him to quickly determine the root cause of performance bottlenecks.

The rest of this paper is organized as follows: section 2 gives a brief overview of profilers and the Cyclops64 architecture; section 3 gives an overview of C64prof; section 4 presents data regarding the overhead from using C64prof; section 5 presents data on the profiler's ease of use as measured by line count changes in the application source code and makefiles; section 6 describes C64prof's visualizers, which are designed to digest tracefiles into graphic representations which are easier for the programmer to understand; section 7 gives conclusions of the data presented in this paper; and section 8 describes future work to be done with C64prof.

2 Background

2.1 Profiling

As high performance architectures have become larger and increasingly complex, program execution has correspondingly become more opaque. The larger the machine and more complex the programming environment, the harder it is for programmers to anticipate bottlenecks and performance maladies. To address this difficulty, a number of performance measurement and analysis tools have been developed.

Profilers often differ in how they select which points in program execution shall be instrumented (manually by the programmer, compiler-selected, event-triggered, etc); in how the program itself is instrumented, if at all (by the compiler during compilation time, by a source-to-source translator, by a binary-to-binary translator, etc); and in the purpose for which the profiler is being used. Frequently, these programs output data files pertaining to program execution, known as traces, which can be parsed and analyzed by other performance analysis tools.

Table 1 gives a summary of some of these tools.

The Cyclops64 profiler was motivated by experiences in the Parallel Architecture Laboratory (PAL), formerly of Los Alamos National Laboratory and now located at Pacific Northwest National Laboratory. The PAL group had developed a tool in house, the PAL profiler, which served as the inspiration for C64prof. The PAL Profiler was intended for use on large distributed memory clusters running MPI applications. It's advantages, versus standard off-the-shelf profilers such as Tau, are fourfold:

- The PAL profiler is smaller, making it easier to port from one system to another
- The PAL profiler is simpler, meaning that there are fewer ways it can degrade the target program's performance

Table 1: Common (dynamic) performance measurement and analysis tools

Name	Instrumentation type	Year	Description
Prof [19]	Compile-time	1979	Uses interrupt-based alarm clock and program counter sampling to measure runtime for functions within a program
Gprof [10]	Compile-time	1982	Uses prof to gather program execution statistics, but augments it by generating call-graphs
Purify [12]	Compile-time	1992	Instruments memory accesses to detect invalid ones and memory leaks.
ATOM [6]	Binary compilation	1994	General purpose - inserts programmer-defined functions into target program.
TAU[1]	Compile-time, runtime	1994	General purpose - inserts programmer-defined functions into target program at source, or during compile-time or runtime.
QPT [13]	Compile-time	1994	Counts basic blocks in target program
Eraser [18]	Binary compilation	1997	Implemented using ATOM. Detects data races arising from incorrect use of locks for shared variables
PAPI[2]	None	1999	Cross-platform API for accessing CPU event counters
OProfile[17]	Runtime	2000	Uses kernel routines to sample the program counter and CPU event counters
DIDUCE [11]	Compile-time	2002	Dynamically determines program invariants and exceptions thereto
Ccured [15]	Compile-time	2002	Uses run-time checking to enforce type safety in C programs
Valgrind [16]	JIT binary translation	2003	General purpose - Inserts programmer-defined functions into target program.
DynamoRio [3]	Binary compilation	2004	General purpose - Inserts programmer-defined functions into target program.
PIN [14]	JIT binary translation	2005	General purpose - Inserts programmer-defined functions. Execution occurs on a PIN VM.

- The PAL profiler, owing to the fact that it was designed in house, was better understood by the PAL team in terms of how it operated and impacted the instrumented program's behavior.
- Owing to all of the above, the PAL profiler is easier to modify.

C64prof started as a black-box reimplementation of the PAL profiler targeted towards the Cyclops64 architecture.† C64prof soon evolved, gaining features that are specific to the Cyclops64 architecture and software ecosystem.

2.2 The C64 architecture

The Cyclops64 (C64) architecture is designed to serve as a dedicated petaflop compute engine for running high performance applications. A complete C64 system consists of up to 13,824 processing nodes, arranged in a 3D-mesh network. Each processing node consists of a C64 chip, external DRAM, and a small amount of external interface logic. A C64 chip employs a many-core-on-a-chip design with 80

†Black box in the sense that the internal workings of the PAL profiler were never examined.

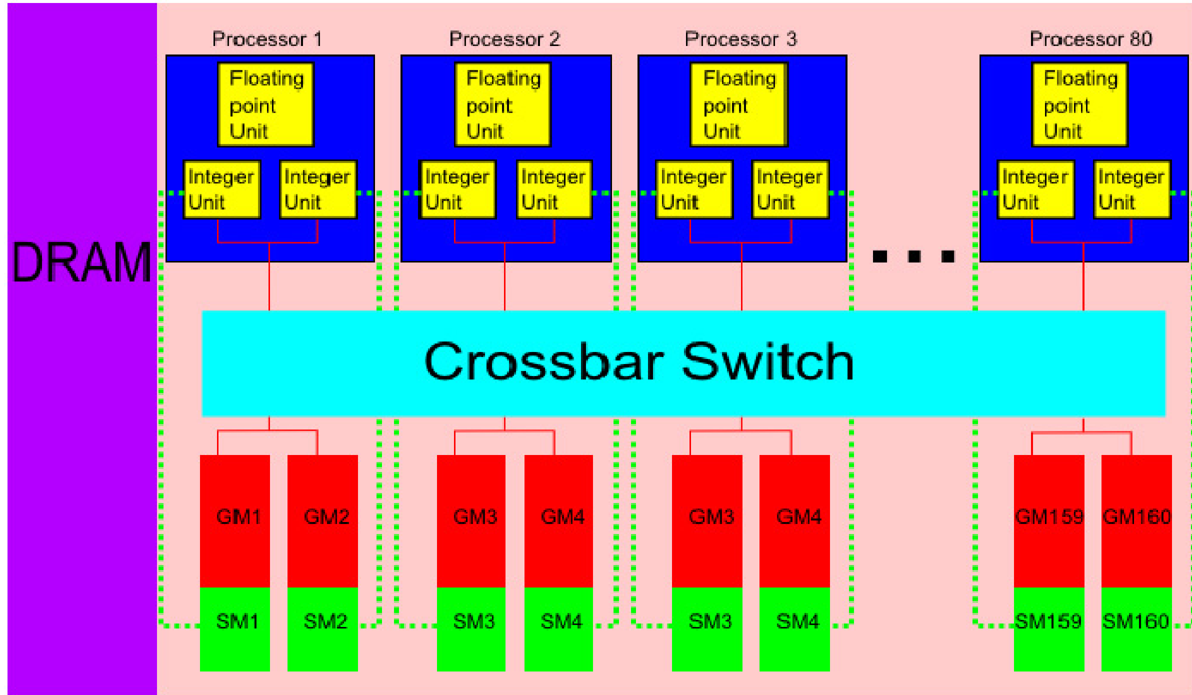

Figure 1: A logical view of the Cyclops64 architecture

processors. Each processor includes two thread units, one floating-point unit and two 32KB SRAM memory banks. 32KB instruction caches are shared by five processors each. An interface to the off-chip DDR2 SDRAM memory and bidirectional inter-chip routing ports completes the design. The C64 chip has no data cache. Instead a portion of each SRAM bank can be configured as scratchpad memory (SP). The remaining sections of SRAM combined form the global memory (GM) that is uniformly addressable from all thread units. On-chip resources are connected by a 96-port crossbar network, which provides a 4GB/s bandwidth per port, in total 384GB/s on each direction. This huge bandwidth sustains all the intra-chip traffic communication and the six routing ports that connect each C64 chip to its nearest neighbors in the 3D-mesh network. [4] [8]

The processors and memories are lain out in "dancehall" fashion, with processors on one side of the crossbar switch and general memories on the other. In addition to connecting to the general memories through the crossbar switch each processor also has a low-latency access pathway to its own scratch pad memory (shown in figure 1 with dashed lines). Each processor has two integer execution units, which share a single float-floating point unit.

The C64 architecture represents a major departure from mainstream microprocessor design in several respects. The C64 chip integrates processing logic, embedded memory and communication hardware in the same piece of silicon. However, it provides no resource virtualization mechanisms. For instance, execution is non-preemptive and there is no hardware virtual memory manager. The former means one single application can run at a given time on a set of C64 nodes and the C64 microkernel will not interrupt the user application unless an exception occurs. The latter means the three-level memory

hierarchy of the C64 chip is visible to the programmer. From the processing core standpoint, a thread unit is a simple 64-bit, single issue, in-order RISC processor with a small instruction set architecture, operating at a moderate clock rate (500MHz). Nonetheless, it incorporates efficient support for thread level execution. For instance, a thread can stop executing instructions for a number of cycles or indefinitely and it can be woken up by another thread through a hardware interrupt. C64 also provides an extremely fast hardware implementation of the barrier synchronization primitive.

3 C64prof

C64prof is a we have profiler created to assist Cyclops64 programmers in analyzing the behavior of their applications and improving their performance. The programmer may specify events to be counted by the profiler using the architecture's two hardware event counters. C64prof uses automated compiler instrumentation to insert calls to monitoring functions at the entry and exit points of functions. These monitoring functions record the event type (which function is being entered or exited, or which SHMEM function was just called), the time at which the event occurred, the thread ID of the host on which it occurred, and the hardware event counts at the time the event occurred.

The gcc compiler contains a flag, -finstrument-functions, that instructs the compiler to automatically add instrumentation calls. To wit: "-finstrument-functions: Generate instrumentation calls for entry and exit to functions. Just after function entry and just before function exit, the following profiling functions will be called with the address of the current function and its call site.

- void __cyg_profile_func_enter (void *this_fn, void *call_site);
- void __cyg_profile_func_exit (void *this_fn, void *call_site);" [7]

The monitoring function parameter this_fn points to the monitoring function itself, and call_site is a pointer to the function that triggered the monitoring function. It is worth noting that in Posix-complaint C, it is not possible to get a function's name from its pointer. However, it is possible using the Glibc functions dladdr or backtrace. The Cyclops64 compiler has a flag, -fcall-stack-profile, that replicates the behavior of gcc's -finstrument-functions identically except for the monitoring function names and prototypes.

3.1 The C64prof API

The C64prof API consists of seven programmer-facing API calls: an initialization function (_noprofile_cygprofile_init) which is used to initialize the profiler and Cyclops64 event counting library; a shutdown function (_noprofile_cygprofile_shutdown) which writes out data and sets the profiler to immediately return from any future instrumentation calls; two manual instrumentation functions (_noprofile_cyclops_profile_begin_highpf and _noprofile_cyclops_profile_end_highpf) which can be called by the programmer in situations where automated instrumentation is undesirable or insufficient; and three functions (_noprofile_set_logfile_size,

_noprofile_set_max_recursion_depth, and _noprofile_get_max_recursion_depth) which allow the programmer to set how much RAM will be allocated to profiler data store, the maximum recursion stack depth after which, instrumentation calls will immediately return, and to query the current recursion stack size.

3.1.1 _noprofile_cygprofile_init

Prototype: int _noprofile_cygprofile_init (int beverbose, int myrank, int e0, int e1)

Description: _noprofile_cygprofile_init initializes the profiler and ECL macros. Following a successful call, the profiler will be initialized, and events will be recorded by the profiler. Although the instrumentation routines may be called prior to the initialization function, they will return without effect until the initialization function is called.

Parameters:

- beverbose - This parameter is deprecated. (Previously, a 1 indicated verbose output, and 0 indicated quiet output). Current versions of the profiler ignore this value.
- myrank - this is an integer with value 0 or greater. Each thread must pass a unique value to this function. This value will be used to name the output file - for example, if one thread calls init with myrank=2, and another thread calls it with myrank=3, their output files will be report-0002.txt and report-0003.txt, respectively. It is recommended that the programmer use tnt_my_pe() for this parameter.
- e0 - This integer specifies which event to count. It must be -1 (to indicate that no event is to be counted) or equal to one of the values defined in the ECL library, which are given here: http://www.capsl.udel.edu/internal/cyclops-64/online-manual/release-2.3/C64ProgrammingManual.htm#_Toc177
- e1 - This integer specifies which event to count. It must be -1 (to indicate that no event is to be counted) or equal to one of the values defined in the ECL library, which are given here: http://www.capsl.udel.edu/internal/cyclops-64/online-manual/release-2.3/C64ProgrammingManual.htm#_Toc177

Return values:

- 0 - if no error occurred
- 1 - an error occurred while attempting to open the output file
- 2 - an error occurred while attempting to set the first hardware event counter (c64ecl_set returned error)
- 3 - an error occurred while attempting to set the second hardware event counter (c64ecl_set returned error)
- 4 - an error occurred while attempting to start counting in the first event counter (c64ecl_start returned error)

- 5 - an error occurred while attempting to start counting in the second event counter (c64ecl_start returned error)
- 6 - an error occurred while allocating RAM for profiler data storage

3.1.2 _noprofile_cygprofile_shutdown

Prototype: void _noprofile_cygprofile_shutdown(void)

Description: _noprofile_cygprofile_shutdown shuts down the profiler and writes all unwritten data to the logfile. No further logging occurs after a successful call.

Parameters: None

Return values: None

3.1.3 _noprofile_cyclops_profile_begin_highpf

Prototype: void cyclops_profile_begin_highpf(uint64_t label)

Description: _noprofile_cyclops_profile_begin_highpf is used by the programmer to manually instrument code. Its effects are identical to cyclops_profile_begin, which is automatically called by the compiler. Labels are used in place of function names.

Parameters:

- label - an unsigned 64 bit integer representing the name of the section of code that is to be entered. Note: in order for the visualizer to work correctly on the output files, every entry label must have a corresponding exit label which is called on a one-for-one basis.

Return values: None

3.1.4 _noprofile_cyclops_profile_end_highpf

Prototype: void cyclops_profile_end_highpf (uint64_t label)

Description: _noprofile_cyclops_profile_end_highpf is used by the programmer to manually instrument code. Its effects are identical to cyclops_profile_end, which is automatically called by the compiler. Labels are used in place of function names.

Parameters:

- label - an unsigned 64 bit integer representing the name of the section of code that is to be exited. Note: in order for the visualizer to work correctly on the output files, every exit label must have a corresponding entry label which is called on a one-for-one basis.

Return values: None

3.1.5 _noprofile_set_logfile_size

Prototype: void _noprofile_set_logfile_size(uint64_t nsize)

Description: Each profiler instance allocates RAM to be used to store the characters which are later written to the logfile. By default, the size of this RAM allocation (the internal variable LOGFILESIZE) is 5 megabytes (= $5 \times 1024 \times 1024$ bytes). The programmer can set this value on a per-profiler thread basis by calling _noprofile_set_logfile_size. The smaller this value is, the less total RAM the profiler will consume; the higher this value is, the less frequently the profiler will flush data to the report files, in turn improving the performance.

Parameters:

- nsize - an unsigned 64 bit integer representing the number of bytes for each instance of the profiler to allocate for character storage prior to writing to the file system. This parameter must be at least 213 (greater than twice MAXENTRYSIZE, the maximum size of each entry in the logfile, which is defined as 111), or the function will fail an assertion and the program will abort.

3.1.6 _noprofile_set_max_recursion_depth

Prototype: void _noprofile_set_max_recursion_depth (uint64_t depth)

Description: _noprofile_set_max_recursion_depth is used by the programmer to limit the number of function calls that will be recorded. This is critically important in programs with large numbers of function calls whose performance would be degraded by excessive instrumentation.

Parameters:

- depth - an unsigned 64 bit integer. If the profiler has observed more function calls than this value, function entries (either compiler triggered or manually triggered using _noprofile_cyclops_profile_begin_highpf) will not be recorded.

Return values: None

3.1.7 _noprofile_get_max_recursion_depth

Prototype: uint64_t _noprofile_get_max_recursion_depth (void)

Description: _noprofile_set_max_recursion_depth is used by the programmer to get the current size of the program stack as observed by the profiler

Parameters: None

Return values: returns the current recursion depth

3.2 PSHMEM - A Profiling Library for SHMEM

PSHMEM is C64prof's profiling wrapper for SHMEM, a library that allows processors in shared-memory architectures to make interrupt-free accesses to other processors' RAM. Each SHMEM call has a matching PSHMEM call, whose name is the same except with _noprofile_ prepended. Both the SHMEM call and its matching PSHMEM call have identical function prototypes.

For example, SHMEM includes this function: void shmem_set_lock (long *lock). Its PSHMEM equivalent is: void _noprofile_shmem_set_lock (long *lock). Note that the number of arguments, argument names, argument types, and return type are identical.

By default, PSHMEM calls produce trace output formatted identically to non-SHMEM function calls. (See the stack trace given above for sample PSHMEM output) However, the 1:1 mapping of PSHMEM to SHMEM calls means that programmers, if they so desire, can modify any or all PSHMEM calls to produce any output they desire.

3.3 Example program and C64prof trace

Consider the following simple C64prof test program (example.c in the C64prof distribution):

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <c64ecl.h>
#include <stdlib.h>
#include <time.h>

#include ‘‘palprofiler.h’’

#include ‘‘cnet.h’’
#include <shmem.h>

int F1(void);
int F2(void);
int F3(void);

int workfunc(int seed){
  int cnt, x, iters, worknum[4];
  srand ( time(NULL) +tnt_my_thread()+seed);

  iters = rand() % 100;
  worknum[0]= rand() % 100;
  worknum[1]= rand() % 100;
  worknum[2]= rand() % 100;
  worknum[3]= rand() % 100;
  for (x=0; x<iters; x++){
    cnt *= worknum[rand()%4];
  }
  return cnt;
}

int F1()
{
```

```
  long val1, val2, val3;

  _noprofile_cyclops_profile_begin_highpf (100);

  val1 = _noprofile__is_spmd();
  val2 = _noprofile_my_pe();
  val3 = _noprofile_num_pes();

  _noprofile_cyclops_profile_end_highpf (100);

  return val1+val2+val3;
}
int F2()
{
  return F1()+workfunc(1)+workfunc(2)+workfunc(3)+workfunc(4);
}
int F3()
{
  return F2()+F1();
}


int main()
{
  int rval;
  int myrank = tnt_my_thread();
  tnt_barrier_t barrier;

  printf(``Rank %d:  Running.\n'', tnt_my_thread());

  rval = _noprofile_cygprofile_init(0, tnt_my_thread(), c64ecl_event_insn, c64ecl_event_load);
  if (rval != 0){
    printf(``Rank %d: Error - cygprofile_init did not run correctly. Returned %d\n'', myrank, rval);
    return rval;
  }


  printf(``Rank %d: Profiler initialized. Initializing SHMEM.\n'', myrank);
  _noprofile_shmem_init();

  printf(``Rank %d: SHMEM initialized. Beginning program execution.\n'', myrank);
  F3();

  printf(``Rank %d: Work terminated. Profiler shutting down\n'', myrank);
  _noprofile_cygprofile_shutdown();


  shmem_finish();
  return 0;
}
```

This program tests the profiler's ability to profile ordinary function calls as well as SHMEM function calls. Workfunc generates a workload that is pseudorandom in both size and nature, meaning that each thread will have profiler traces that are different from each other, and different from themselves if the program is re-run.

When run, the code produced the following trace from the 0th processor:

```
Recorded c64ecl_event_insn,c64ecl_event_load
87563,-1,P_START
179404,-1,E,_noprofile__is_spmd,8641,695
250842,-1,X,_noprofile__is_spmd,39550,497
327124,-1,E,F3,3241,284
337378,-1,E,F2,104,11
347424,-1,E,F1,97,11
359266,-1,E,100,741,72
368981,-1,E,_noprofile__is_spmd,79,9
380284,-1,X,_noprofile__is_spmd,82,8
391532,-1,E,_noprofile__is_spmd,81,9
402838,-1,X,_noprofile__is_spmd,82,8
414143,-1,E,_noprofile__is_spmd,81,9
425444,-1,X,_noprofile__is_spmd,82,8
438660,-1,X,100,743,72
448404,-1,X,F1,97,11
458309,-1,E,workfunc,107,11
475565,-1,X,workfunc,1296,138
486227,-1,E,workfunc,117,13
500943,-1,X,workfunc,861,93
511408,-1,E,workfunc,117,13
525448,-1,X,workfunc,745,81
535912,-1,E,workfunc,117,13
553670,-1,X,workfunc,1383,147
564313,-1,X,F2,109,13
574373,-1,E,F1,107,13
586311,-1,E,100,741,72
596033,-1,E,_noprofile__is_spmd,79,9
607332,-1,X,_noprofile__is_spmd,82,8
618645,-1,E,_noprofile__is_spmd,80,9
629944,-1,X,_noprofile__is_spmd,82,8
641247,-1,E,_noprofile__is_spmd,80,9
652494,-1,X,_noprofile__is_spmd,82,8
665716,-1,X,100,742,72
675456,-1,X,F1,97,11
685413,-1,X,F3,97,11
723622,-1,P_STOP,,2956,265
```

The first column is the time (in cycles since the start of program execution) at which an event occurred. The second column (-1) has no meaning. (It exists because C64prof was originally intended to be backwards compatible with the visualization toolchain created for the PAL profiler at Los Alamos National Laboratory). The third column is the type of event recorded. An “E” denotes a function entry;

an “X” denotes a function exist; “P_START” indicates the initialization function being called; “P_STOP” indicates the shutdown function being called. The fourth column, where one is given, gives the name of the function call being profiled. Names indicate compiler-profiled functions, and numbers indicate labels given by the programmer to manually instrumentation calls. The final two columns, the fifth and sixth, give the number event hardware event counts recorded when those events occurred. The event counters are cleared after every function entry or exit.

The first line in the file is the name of events to be profiled. The event names are derived from labels given to them in the Cyclops64 event counting library. (See that library’s documentation for further details.) In the case of the above trace, the events counted were the number of instructions and loads executed, respectively.

3.4 Interpreting the C64prof trace

When analyzing a program, a programmer will want to know how many of a certain type of event occurred during the execution of a particular function. This can be calculated from the trace files. Consider the following segment from the C64prof trace given in section 3.3:

```
347424,-1,E,F1,97,11
359266,-1,E,100,741,72
368981,-1,E,_noprofile__is_spmd,79,9
380284,-1,X,_noprofile__is_spmd,82,8
391532,-1,E,_noprofile__is_spmd,81,9
402838,-1,X,_noprofile__is_spmd,82,8
414143,-1,E,_noprofile__is_spmd,81,9
425444,-1,X,_noprofile__is_spmd,82,8
438660,-1,X,100,743,72
448404,-1,X,F1,97,11
```

Function F1 calls (manually-instrumented psedu-function) 100, which calls _noprofile__is_spmd multiple times. Diagrammatically, this can be expressed as figure 2.

As the diagram makes clear, events which are executed by some target function will be recorded by the profiler either during the entry (“E”) event of another function called by the target function, or when the target function returns. (In the diagram, these would be the arrows pointing away from the target function, either to its left or right). Mathematically, the target function’s event count (EC) can be expressed as:

$$EC_{exclusive} = \Sigma EC_{@call} + EC_{exit} \quad (1)$$

where $EC_{@call}$ are the event counts recorded when entering functions called by the target function, and EC_{exit} is the event count recorded when the target function exits. This approach is exclusive because it excludes events that occur in functions called by the target function. However, for debugging purposes,

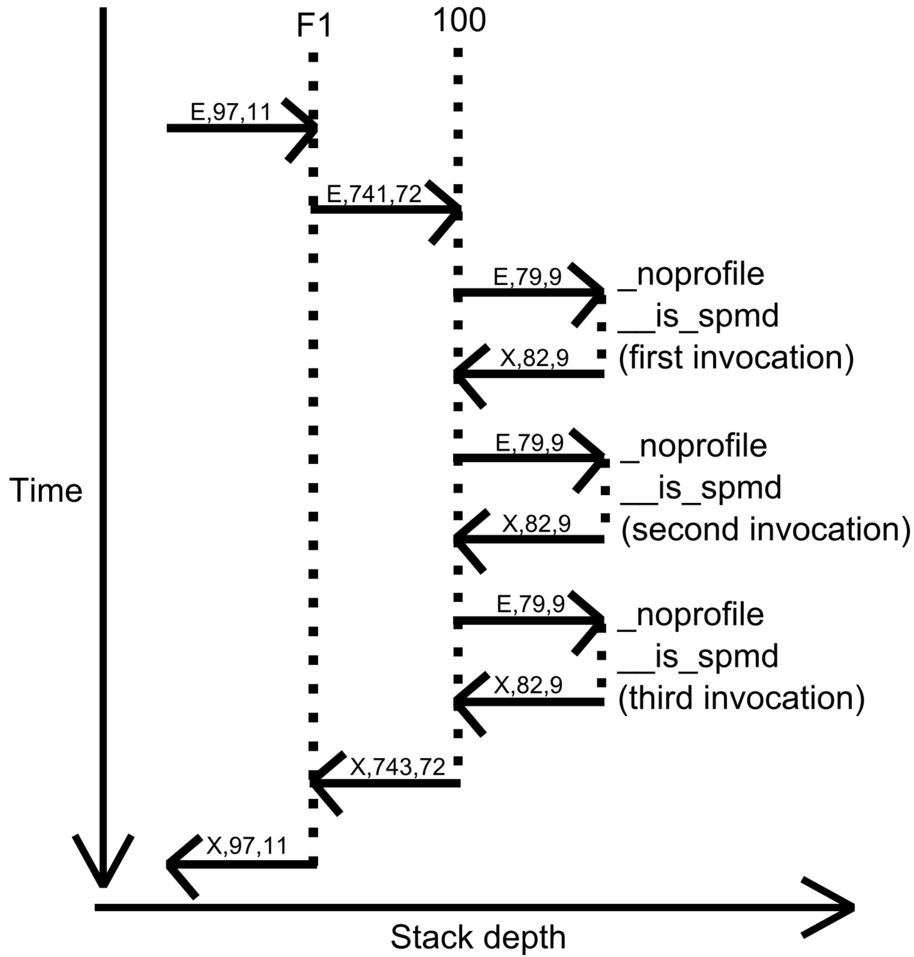

Figure 2: A diagrammatic view of the C64prof trace

it may also be useful to the programmer to include these. This inclusive approach can be expressed mathematically as:

$$EC_{inclusive} = \Sigma EC_{@call} + \Sigma EC_{called} + EC_{exit} \tag{2}$$

where ΣEC_{called} is the sum of all the events that occur in all functions called by the target function.

In figure 2, F1's exclusive event count for the first profiled event would be 741 (the number of events that had occurred at the time it called function 100) + 97 (the value when the function returns); if calculated inclusively, it is 741+79+82+79+82+79+82+743+97. Both inclusive and exclusive approaches have performance analysis value to the programmer.

4 Runtime Overhead

When using automated instrumentation, C64prof's overhead occurs at function calls. Therefore C64prof's runtime factor increase (the factor by which program runtime is increased when using C64prof) will be

directly proportional to the granularity of the function calls. A program with lots of work in few function calls will experience little overhead; conversely, a program with many fine-grained function calls will experience a correspondingly high overhead.

In order to evaluate the runtime overhead from using the profiler, we ran instrumented versus uninstrumented comparative tests using two scientific applications - Matrix-Matrix multiply and Finite-difference time-domain method - and most of the benchmarks in the Splash2 suite. The Splash2 benchmarks were implemented using their respective default problem sizes.†

Data was collected using the Cyclops64 functionally accurate simulator. The cycles per event was calculated by taking the profiled runtime, subtracting the unprofiled runtime, and dividing by the number of profiler-recorded events.

4.1 Splash2 benchmark suite

Table 2: C64prof overhead on the Splash2 benchmarks

Benchmark	Unprofiled Runtime (Cycles)	Profiled Runtime (Cycles)	Runtime increase factor	Event count	Cycles per event
Barnes-Hutt	30592638	72671602	2.38	131161 †	320.82 †
Ocean (Contiguous partitions)	66041	86154	1.30	1708	11.78
Ocean (Non-Contiguous partitions)	62052	88604	1.43	1842	14.41
Water (Spatial)	14052219	14052721	1.00	19	26.42
Water (N-Squared)	10649200	10649590	1.00	19	20.53
FMM	288997	795188	2.75	32659	15.50

4.2 Finite-difference time-domain method

The Finite-difference time-domain method (FTDT) application we profiled was written by Daniel Orozco in C. It tests various tiling approaches to FDTD. We profiled the application using a real-world problem size.

Table 3: C64prof overhead on the FTDT

Threads	Unprofiled runtime (Cycles)	Profiled Runtime (Cycles)	Runtime Increase factor	Events	Cycles per event
19	956101903	956474437	1.0003	534	697.63
127	146238833	147576976	1.009	3738	357.98

†: In order to run in a reasonable time on the simulator, a stack limit of 4 was used on Barnes-Hutt simulations. Therefore, the number of counted events is only a tiny fraction of the actual number of events, which was 99,977,608. Events which were below the stack limit triggered an entry into an instrumentation function call, followed by an if statement and then a return, thus slightly increasing the program runtime but not as much as a full recording.

4.3 Assembly-coded matrix-matrix multiply

Cyclops64 has been the target of successful attempts to implement high-performance matrix-matrix multiply, as described in [9]. We used this implementation as the basis for experiments testing the profiler with high-performance code. The computation in this code is hand-written in highly-optimized assembly, allowing it to achieve more than half of Rmax, the maximum theoretical performance of the Cyclops64 chip. This highly optimized code which has no function calls for the compiler to instrument presents an interest worst-case scenario for the profiler both performance-wise and usability-wise.

We profiled the code using the manual instrumentation functions, _noprofile_cyclops_profile_begin_highpf and _noprofile_cyclops_profile_end_highpf (described in sections 3.1.3 and 3.1.4 above) along with the requisite initialization and shutdown calls.

Table 4: C64prof overhead on the Matrix-matrix multiply

Threads	Unprofiled runtime (Cycles)	Profiled Runtime (Cycles)	Runtime Increase factor	Event count	Cycles per event
4	2709778	9657752	3.56	1160	5989.63
16	717702	4945521	6.89	1184	3570.79
64	583495	3705473	6.35	1280	2439.05
100	1150038	2786087	2.42	1000	1636.05
144	2018043	3809037	1.88	1440	1243.75

The irregularity of the overhead here is the result of using an increasing number of processors to process a fixed-size matrix-matrix multiply, which results in a stair-stepping distribution of work. (This can also be seen by dividing the number of events by the number of threads.)

5 Ease of Use

Ease of use is an important quality any piece of software, though notoriously difficult to quantify. In the case of C64prof, lines of code added or changed (LOC) in order to use the profiler is an adequate though imperfect proxy measurement of C64prof's ease of use. Table 5 gives the LOC for each splash benchmark, and that benchmark's accompanying Makefile.

Table 5: Lines of code add/modified in order to get C64prof to profile the Splash benchmarks

	Code	Makefile
Barnes	5	16
Ocean (Contingious partions)	4	5
Ocean (Non-contingious partions)	4	5
Water-spatial	4	29
Water-nsquared	4	29
FMM	4	5

6 C64prof Data Visualization

C64prof includes a powerful visualizer. The visualizer parses C64prof report files to generate one or more GNUplot command files, and then invokes GNUplot on the command files to generate images. The visualizer operates in single or multiple-pass mode. In single-pass mode, traces are analyzed for a single run of a profiled application. In multi-pass mode, traces from multiple runs are analyzed. Those runs must be logically and programmatically identical except for the hardware events being counted.

6.1 Single-pass mode

In single-pass mode, the visualizer generates an image showing the timing synchronization between various threads and the functions they invoke (figure 3), and one image for each type of hardware event being counted (figure 4)

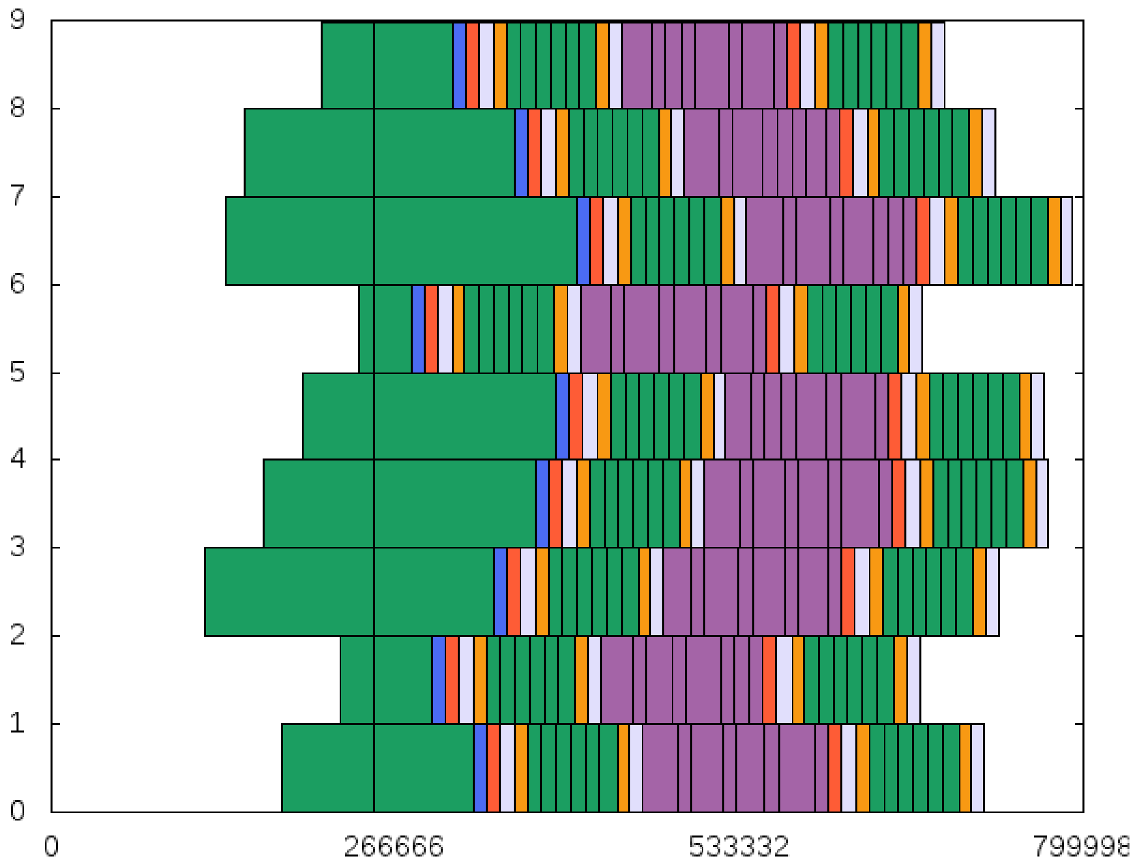

Figure 3: The C64prof visualizer in single-event mode synchronization graph for example.c

Figure 3 gives the single-pass mode output from the visualizer using the report files generated from example.c (described above in section 3.3). The X-axis represents the time (in cycles), and the Y-axis

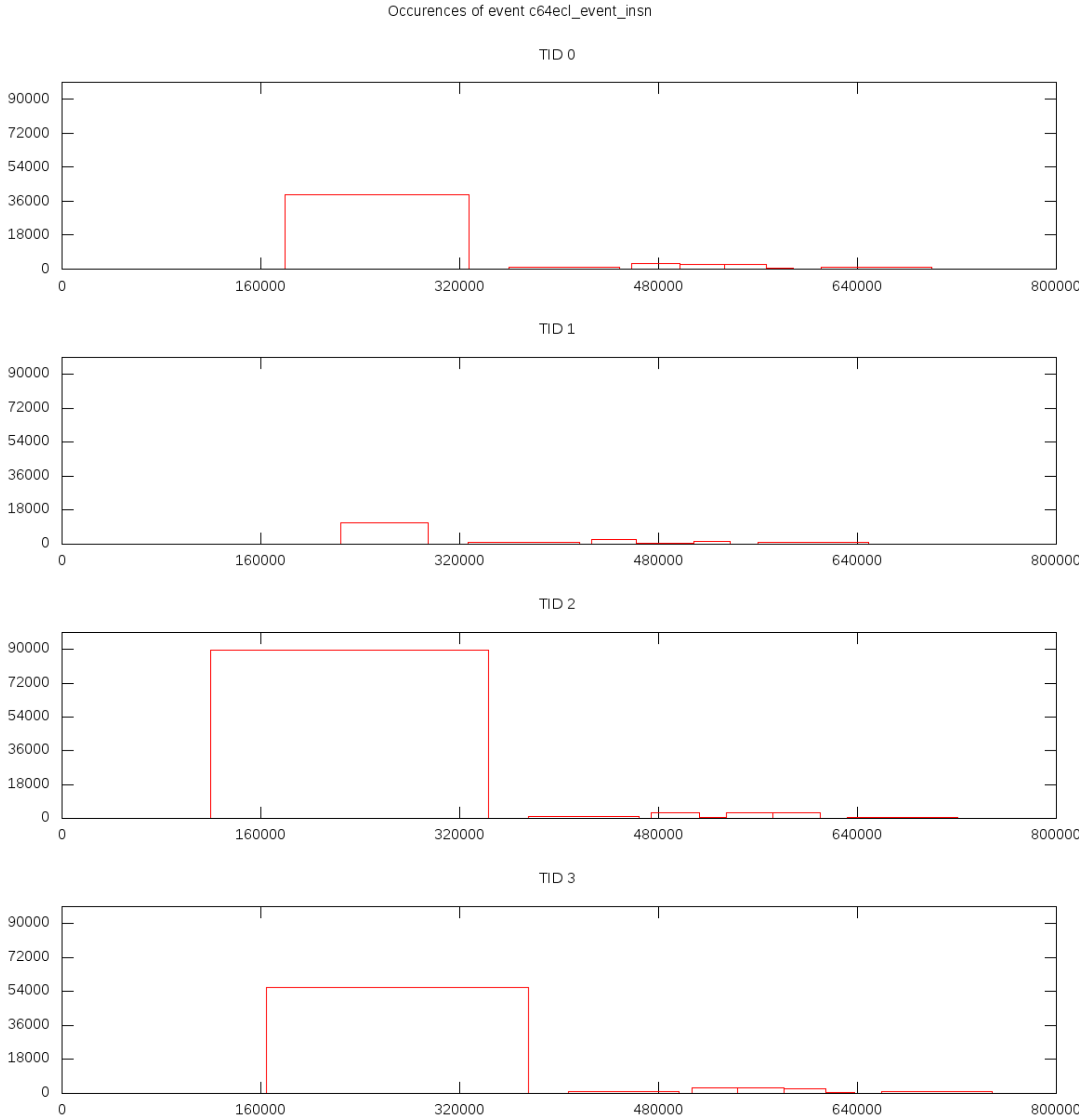

Figure 4: The C64prof visualizer event count graph generated from example.c trace data (cropped view)

represents the profiler rank, which is passed by the programmer during the profiler initialization call and typically corresponds to the thread ID. Each rectangle represents a function call on a given host. Rectangles are deterministically colored according to the name of the function they represent.†

If more specific information is needed, the programmer can get the function names and precise start

†Specifically, the color is given by a hex triplet which is taken from the first six characters of the md5 hash of the function name. For example, the name "workfunc" hashes to a464a75d42e21723adcd4987d3446e39. Interpreted as a hex triplet, the first six characters (a464a7) are the purple rectangles given in figure 3

and end cycles from the command file itself, which is formatted like so:

```
set object 1 rect from 179404,0 to 250842,1 fc rgb "#1b9e61" # _noprofile__is_spmd
set object 2 rect from 250842,0 to 327124,1 fc rgb "#1b9e61" # _noprofile__is_spmd
set object 3 rect from 327124,0 to 337378,1 fc rgb "#4b6bf4" # F3
set object 4 rect from 337378,0 to 347424,1 fc rgb "#fe5c36" # F2
set object 5 rect from 347424,0 to 359266,1 fc rgb "#e1dffc" # F1
set object 6 rect from 359266,0 to 368981,1 fc rgb "#f89913" # 100
set object 7 rect from 368981,0 to 380284,1 fc rgb "#1b9e61" # _noprofile__is_spmd
```

The final column in each entry gives the function name; the penultimate column gives the function color (as a hex triplet); and the coordinates (6th and 8th columns) give the position of the lower-left and upper-right corner of each rectangle.

6.2 Multi-pass mode

Figure 5 shows the C64prof multievent graph for example.c. The program was run four times gathering two types of events each, for a total of eight profiled events: instructions executed, loads, SPM loads, SPM stores, SRAM loads, SRAM stores, DRAM loads, and DRAM stores. The graph shows, for each host, the total number of each type of event that occured, colored by function. At a glance, the programmer can see for each host which type of event dominates, as well as which function(s) dominates each event type.

7 Conclusions

With C64prof, we have achieved our goal of providing Cyclops64 programmers with a powerful, easy-to-use performance analysis tool. We have shown, with our experience using the profiler to analyze the matrix-matrix multiply, FDTD, and Splash2 benchmark codes, as well as our LOC ease-of-use metric, that the profiler is easy use. The profiler runtime overhead measurements show that the profiler overhead is very low for long running code (the FDTD) and reasonably low (with consistently low per-event overhead) on the short-running Splash2 benchmarks.

Even in the extreme case of the hand-coded near-optimal matrix-matrix multiply with no function calls to serve as points of instrumentation, the manual instrumention routines provided the “calipers” to allow the programmer to specify which part of the program to instrument, albeit with substantial overhead.

The visualization extension to the profiler give the programmer a quick way to digest complex tracefiles, and serves as a platform for programmer-created custom visualization tools.

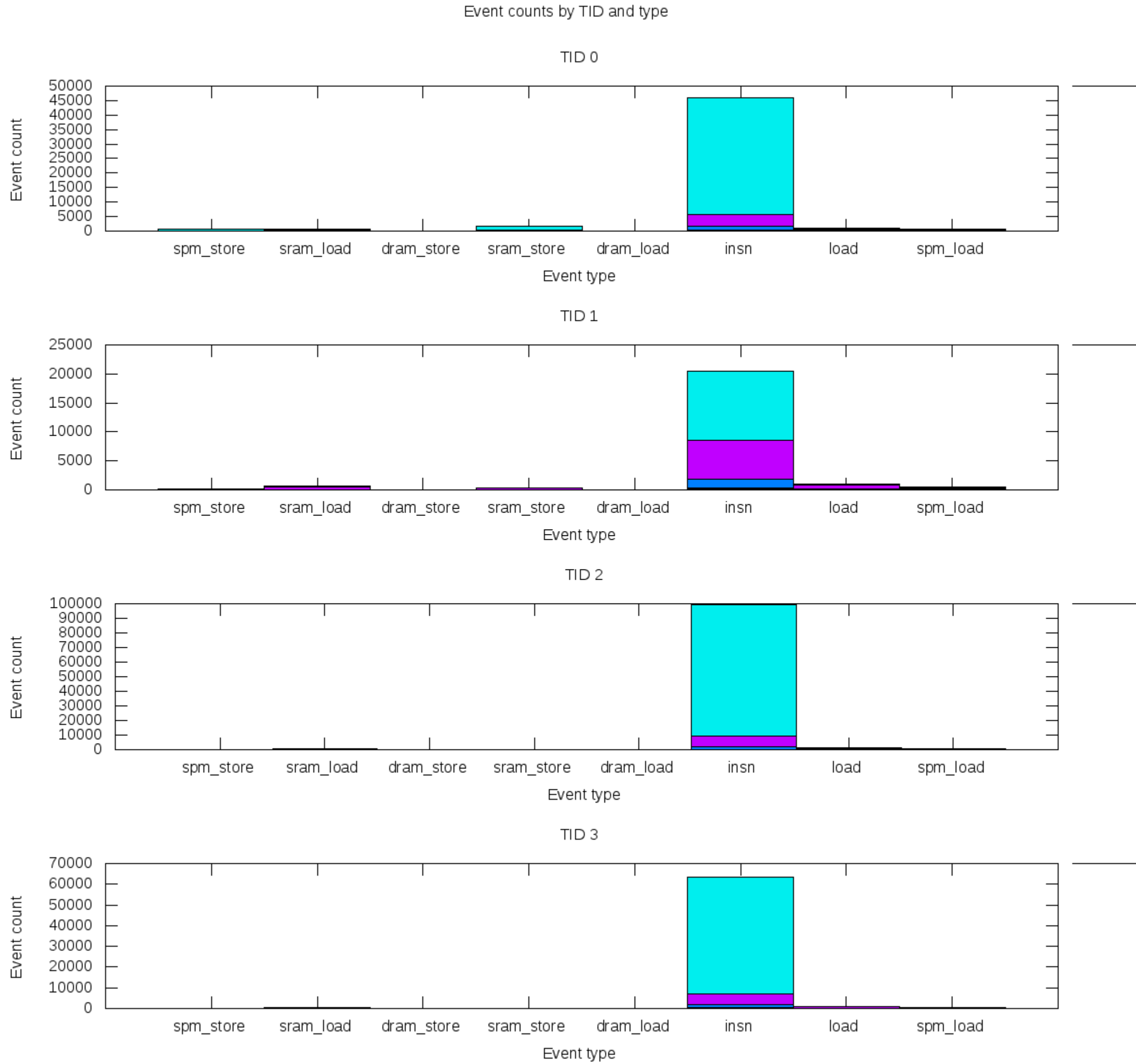

Figure 5: The C64prof visualizer multi-event graph for example.c (cropped view)

8 Future Work

Future C64prof work will be focused on two areas: general codebase maintence and the Fresh Breeze case study. General maintainence involves bug fixes and feature additions (preferably demand-driven). Access to real Cyclops64 hardware is sure to stimulate additional development in both categories.

The Fresh Breeze case study compromises using C64prof to instrument Cyclops64 impementations of the Fresh Breeze execution model. [5]

Acknowledgements

This work was made possible with the support of the members of the Performance Architecture Laboratory, formerly of Los Alamos National Labarotary and now at Pacific Northwest National Laboratory. In particular, we wish to thank Darren Kerbyson, Kevin Barker, and Adolfie Hoisie.

We'd also like to thank Jean-Christophe Beyler, of ET International, who created Cyclop64's automated compiler instrumentation feature, which is essential to the functioning of this profiler.

Profiling on the Matrix-Matrix multiply and FDTD applications would not have been possible if not for the assistance provided by CAPSL members Daniel Orozco and Elkin Garcia. Daniel and Chen Chen, also of CAPSL, provided superlative assistance in implementing the profiler. We are greatly in their debt.

References

[1] Darryl I. Brown, Steven T. Hackstadt, Allen D. Malony, and Bernd Mohr. Program Analysis Environments for Parallel Language Systems: The TAU Environment. *Proceedings of the Workshop on Environments and Tools For Parallel Scientific Computing*, May 1994.

[2] S. Browne, C. Deane, G. Ho, and P. Mucci. PAPI: A Portable Interface to Hardware Performance Counters. In *Proceedings of Department of Defense HPCMP Users Group Conference*, June 1999.

[3] D. Bruening, T. Garnett, and S. Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. *1st International Symposium on Code Generation and Optimization (CGO-03)*, March 2003.

[4] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. Toward a software infrastructure for the cyclops-64 cellular architecture. *hpcs*, 0:9, 2006.

[5] Jack B. Dennis. The Fresh Breeze Model of Thread Execution. `http://csg.csail.mit.edu/Users/dennis/pmup-final.pdf`.

[6] Alan Eustace and Amitabh Srivastava. ATOM: A Flexible Interface for Building High Performance Program Analysis Tools. In *USENIX Winter*, pages 303–314, 1995.

[7] Free Software Foundation. *GCC 4.32 Manual*. Chapter 3.18 - Options for Code Generation Conventions. "`http://gcc.gnu.org/onlinedocs/gcc-4.3.2/gcc/Code-Gen-Options.html`".

[8] Guang R. Gao. Programming and Compiling for TiNy Threads (TNT) – Experience with Cyclops-64 Architecture, December 2006.

[9] Elkin Garcia, Ioannis E. Venetis, Rishi Khan, and Guang R. Gao. Optimized Dense Matrix Multiplication on a Many-Core Architecture. In *Proceedings of International European Conference on Parallel and Distributed Computing (Euro-Par'10)*, 2010.

[10] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a Call Graph Execution Profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[11] Sudheendra Hangal and Monica S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *Proceedings of the International Conference on Software Engineering*, May 2002.

[12] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter 1992 USENIX Conference*, pages 125–138, San Francisco, California, 1991.

[13] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. Technical Report CS-TR-92-1083, Madison, WI, USA, 25 March 1992.

[14] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200. ACM Press, 2005.

[15] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[16] Nicholas Nethercote and Julian Seward. Valgrind: A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[17] OProfile - A System Profiler for Linux. `http://oprofile.sourceforge.net/news/`.

[18] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[19] Ken Thompson. *Unix Programmer's Manual*. Bell Laboratories, Murray Hill, NJ, 7th edition, January 1979. "`http://cm.bell-labs.com/7thEdMan/v7vol1.pdf`", page 134.