USING DYNAMIC BINARY INSTRUMENTATION TO CREATE FASTER, VALIDATED, MULTI-CORE SIMULATIONS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Vincent Michael Weaver

May 2010

© 2010 Vincent Michael Weaver

ALL RIGHTS RESERVED

USING DYNAMIC BINARY INSTRUMENTATION TO CREATE FASTER, VALIDATED, MULTI-CORE SIMULATIONS

Vincent Michael Weaver, Ph.D.

Cornell University 2010

The Memory Wall continues to be a problem with modern systems design. While the steady increase in processor speeds has abated somewhat, Moore's Law continues to provide more transistors to chip designers. This leads to an increase in the number of processors and threads located per chip, which increases the demands on memory systems. Current simulation technology is not able to keep up, leading to sacrifices in methodology and accuracy in order to get results in reasonable time.

Because cycle-accurate simulators are so slow, various methods for reducing execution time can be used. Unfortunately these methods can introduce variations in results of between 10-50% when compared to full reference input sets. Limitations of academic simulators also constrain the architectures under study, with results generated for obsolete or uninteresting systems.

We analyze the performance and accuracy of various limited-execution methodologies. We investigate how deterministic execution affects the measurement of error. We then evaluate using Dynamic Binary Instrumentation (DBI) as an alternative to cycle-accurate simulation. We compare our results to actual systems using hardware performance counters. We look first at a simple 32-bit RISC system, and then look at more complex 64-bit x86 based systems. Finally we investigate the feasibility of using the same methodology for modern multi-processors simulations.

BIOGRAPHICAL SKETCH

Vincent Weaver was born in 1978 and grew up in Joppatowne, Maryland. He attended Joppatowne Elementary and Magnolia Middle schools before moving on to The John Carroll School. He received his B.S. in Electrical Engineering from the University of Maryland College Park in December of 2000. After graduation he briefly worked at Frontpath, a maker of tablet PCs located in Billerica Massachusetts. The dot-com bust caught up with the company, and after a round of layoffs Vince returned to Maryland and worked as a contractor for the U.S. Army creating web front-ends for legacy Fortran applications. In the Fall of 2003 he entered the M.S./Ph.D. program at Cornell University. He obtained a M.S. degree in Electrical and Computer Engineering from Cornell in January of 2009 and his Ph.D. in May of 2010. Vince is a Linux enthusiast who is often accompanied by guinea pigs. He enjoys retro-computing and can program in over 20 types of assembly language.

To Kristina and Elena, for their unfailing support.

ACKNOWLEDGMENTS

First and foremost I would like to thank my advisor, Sally McKee, for her leadership and guidance throughout my time in grad school. Without her help and support none of this would have been possible. Most notable is her amazing ability to accumulate computing clusters, without which this work would have not been finished in a reasonable amount of time. I would like to thank my other committee members, Rajit Manohar and David Albonesi for their insights and feedback that have been instrumental in improving this thesis. In addition I would like to thank Bruce Jacob from the University of Maryland. It was his computer organization and computer architecture classes that set me on the path that resulted in this research.

I also would like to thank all of the members of the Fusion group, past and present, for all their help and support. This includes Martin, Pete, Brian, Chris, Cat, Karan, and Major as well as many others who were not around as long but were just as important.

This work was helped by many open source software projects. I would like to thank the developers of the Linux kernel, especially Linus Torvalds. I would like to thank the perfmon2 developers, especially Stéphane Eranian, as well as the developers of Qemu, Valgrind, and m5.

Additional thanks to the Intel Corporation for donating processors which are used in our Sampaka and Domori clusters.

Part of this work is supported by the National Science Foundation under Grants 0509406 and 0325536 as well as by NSF CCF Award 0702616 and NSF ST-HEC Award 0444413.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

We investigate various methods of speeding up computer architectural simulation, validating the results against real hardware using performance counters. We evaluate RISC, CISC, and CMP-CISC systems. We find that a Dynamic Binary Instrumentation (DBI) based simulation methodology improves run-time over cycle-accurate simulation by at least an order of magnitude, enabling more complete results using full input sets.

Our primary motivation is the Memory Wall [162], which notes that modern system performance is held back by the speed of the memory system. While the steady increase in processor speeds has abated somewhat, Moore's Law continues to provide more transistors to chip designers. This leads to an increase in the number of processors and threads located per chip, which increases the demands on already overloaded memory systems.

In order to address the Memory Wall and other performance problems, the underlying architecture must be studied in detail. The only practical way to do this is with simulators, usually fully in software, that simulate various parts of a computer. As systems get more complicated, simulators get larger, slower, and harder to understand. With the decline of RISC processors and the rise of the Intel x86 architecture, it has become increasingly difficult to create relevant cycle-accurate simulators. Each additional generation of features slow simulators further; development time is spent enhancing micro-architectural simulation and not tuning for speed. Often external effects such as I/O and DRAM are not investigated as thoroughly as internal effects. This is disappointing, as

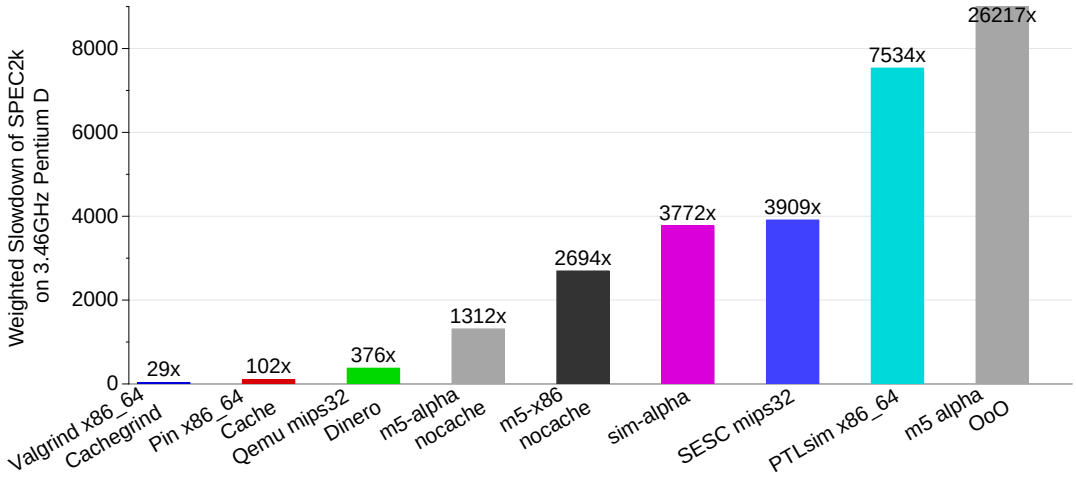

Figure 1.1: Weighted slowdowns of various simulators when running SPEC CPU2000

these externalities are critical to overall performance.

Simulation speed is critical in architectural research. Unfortunately even the fastest DBI methods slow execution by almost a factor of 30, and cycle-accurate simulators slow execution by over a factor of 100 (see Figure 1.1 for slowdowns from various common academic simulation methods). These slowdowns are enough to make a minutes long benchmark take over a day to execute. Some simulators, especially those of complex out-of-order processors, can slow execution by over 1000 times, making single simulations take weeks to months. This drastically increases the hardware required for experiments, as large clusters of computers are required to run simulations in parallel to mitigate long execution times. Simulator bugs become difficult to find, as it might take days to reproduce problems, leading to inefficient debugging sessions. Validation against real hardware also suffers, as proper validation requires many iterations of runs to fine-tune simulator parameters.

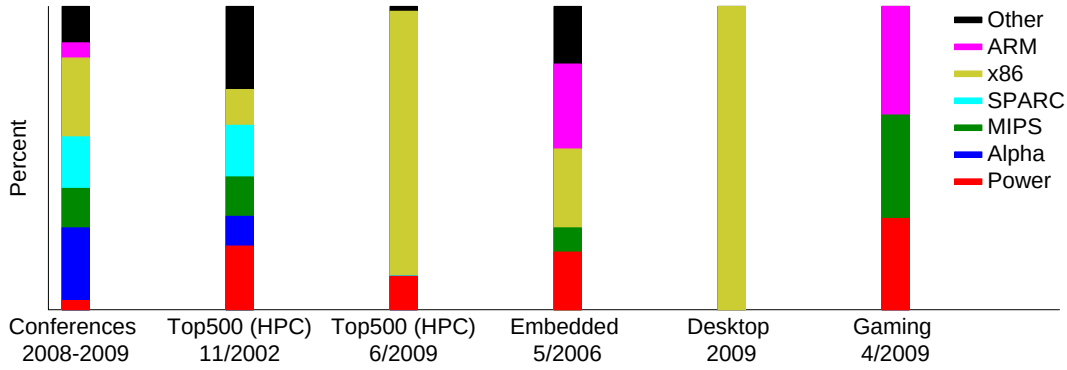

Figure 1.2: Instruction set diversity across various domains. Recent computer architecture conference papers (ICCD'09, ISCA'09, MICRO'08 and ASPLOS'09) match years-old high-performance computing diversity rather than modern trends in computing

Because cycle-accurate simulators are so slow, various methods have been proposed for reducing execution time. Unfortunately these methods can introduce differences in results between 10-50% when compared to full reference input sets [168]. We investigate the effectiveness of many of these methods in Chapter 3. The reduced methods involve simulating small amounts of code, often only a few hundred million to a few billion instructions. On a modern multi-gigahertz chip this equates to less than a second of run time; the reduced execution can miss longer timescale events such as disk and network I/O, operating system context-switches, thermal events, etc. To allow for longer-running input sets, we evaluate using faster Dynamic Binary Instrumentation (DBI) methods of execution.

Another current limitation is the lack of accurate and up to date academic simulators. Once available simulators become "good enough" there is little incentive to do more than incremental improvements. The computer industry

moves quickly and simulators cannot keep pace. This leads to simulators supporting older, simpler, architectures. As shown in Figure 1.2, the mix of architectures simulated in papers from recent conferences (ICCD'09, ISCA'09, MICRO'08 and ASPLOS'09) is unlike any current workloads (gaming [1], embedded [4], desktop, or high performance computing [5]); in fact the most similar workload we find is Top 500 list [5] from seven years ago. This dependence on older architectures makes it difficult to determine if suggested improvements apply to current implementations. Speedups for obsolete systems might not be relevant to the vastly different chips being produced today. We attempt to avoid these limitations by investigating current 64-bit x86 architectures in addition to a more traditional RISC MIPS simulation environment.

One final barrier to accurate simulation is the recent proliferation of multi-core machines. Processor designers are limited by how much performance they can squeeze out of a design before thermal and power issues make the design infeasible. Moore's law of ever increasing transistor counts still holds, but the most common solution is to place more cores per chip. Most non-embedded processors have at least two cores, if not more, per package. This is unfortunate, as adding cores complicates simulators and makes them slower. Most simulators are single-threaded themselves, and can only simulate multiple cores by interleaving execution. This causes a linear increase in slowdowns as more cores are added, compounding the already critical slowness in simulations. In addition, many of the techniques used to improve single-core simulation either do not work for CMP or else are not thoroughly tested enough to know accuracy tradeoffs. We undertake preliminary examinations to address the CMP problem in regard to DBI-based simulation.

CHAPTER 2

RELATED WORK

Slow simulators are an ever-present limitation, holding back the work of computer architects. There is much work attempting to mitigate the problem, some more successful than others. The problem has been attacked from many angles, and there are many related topics that also must be investigated. Our work encompasses many of these different areas, requiring comparison to a large body of related work.

2.1 Reduced Execution Validations

One way to speed simulations is to simulate smaller workloads. Yi et al. [168, 169, 170] investigate the six most common ways of workload reduction: representative sampling (SimPoint [132]), statistics based sampling (SMARTS [163]), reduced input sets (SPEC training inputs, MinneSPEC [77]), simulating the first X Million instructions, fast-forwarding Y Million instructions and simulating X Million, and fast-forwarding Y Million, performing architectural warmup, then simulating X Million. They conclude that SimPoint and SMARTS give the most accurate results. We find similar results, although we do not investigate statistics-based sampling. Their work uses the WATTCH [28] simulator to characterize their results using ten SPEC CPU2000 benchmarks; we use hardware performance counters and the complete CPU2000 and 2006 benchmarks while exploring more architectures and compilers.

Another method of reducing run time is to avoid running redundant benchmarks. Phansalkar et al. [122] use various techniques to determine redundancy

in the SPEC CPU2006 suite and propose eliminating the need to run all of the benchmarks. They use Pin, as well as hardware performance counters on four different architectures (Power, SPARC, itanium, x86) and find that 6 of 12 integer and 8 of 17 FP benchmarks can capture most of the overall benchmark behavior.

Eeckhout et al. [49] propose a hybrid method of reduced inputs and sampling, determining in advance which method works best on a benchmark-by-benchmark basis.

There have been attempts to speed simulation by moving to a hardware based approach. Chiou et al. [33, 32] propose FAST, which is a timing simulator implemented in an FPGA. This speeds simulation by having the slow timing simulation implemented in fast hardware. The Qemu tool is used to generate traces which are fed into the timing simulator; Qemu is modified to handle wrong-path execution. Qemu also handles correctness and operating system issues. This work is not validated, and its speed is limited by the Qemu trace generation.

2.2 SimPoint Validation

There have been many papers published that investigate the SimPoint methodology; our work encompasses more architectures and more implementations than any previous work. We also explore in detail the speed and accuracy of Basic Block Vector (BBV) generation, which is a critical step in undertaking SimPoint analysis (an extension of our HiPEAC work [155]). BBV generation is not discussed in depth in previous papers. We validate our results with hardware performance counters, whereas most previous work validates solely using sim-

ulation.

Sherwood, Perelman, and Calder [131] introduce the SimPoint methodology, which uses basic block distribution to investigate phase behavior. They use SimpleScalar [30] to generate the BBVs, as well as to evaluate the results for the Alpha architecture. They show preliminary results for three of the SPEC95 benchmarks and three of the SPEC CPU2000 benchmarks. They build on this work and introduce the original SimPoint tool [132]. They use ATOM [135] to collect the BBVs and SimpleScalar to evaluate the results for the SPEC CPU2000 benchmark suite. They use an interval of 10M instructions, and find an average 18% IPC error for using one simulation point for each benchmark, and 3% IPC error using between 6 to 10 simulation points. These results roughly match ours.

Perelman, Hamerly and Calder [118] investigate finding "early" simulation points that can minimize fast-forwarding in the simulator. We do not investigate early points as that functionality is no longer available in current versions of the SimPoint tool. When looking at a configuration similar to ours, with 43 of the SPEC2000 reference input combinations, 100M instruction intervals, and up to 10 simulations per benchmark, they find an average CPI error of 2.6%. This is better than what we find using performance counters. They collect BBVs and evaluate results with SimpleScalar, showing that the results on one architectural configuration track the results on other configurations while using the same simulation points. We also find this to be true when comparing different implementations of the same instruction set architecture.

When reporting results that use the SimPoint methodology, often no mention is made of how the underlying BBV files are collected. If not specified, it is usually assumed that the original method described by Sherwood et al. [132]

is used, which involves ATOM [135] or SimpleScalar [30]. The SimPoint website provides pre-generated simulation points for a set of Alpha SPEC CPU2000 binaries; the use of these makes gathering BBV files unnecessary. Some works mention BBV generation briefly, with no indication of any validation. For example, Nagpurkar and Krintz [107] implement BBV collection in a modified Java Virtual Machine in order to analyze Java phase behavior, but do not specify the accuracy of the resulting phase detection.

Patil et al.'s PinPoints [115] use the Pin [87] tool to gather BBVs, and then validate the results on the Itanium architecture using performance counters. This work predates the existence of Pin for x86, so no x86 results are shown. Their results show that 95% of the SPEC CPU2000 benchmarks have under 8% CPI error when using up to ten 250M instruction intervals. All their benchmarks complete with under 12% error, which is more accurate than our results. This is potentially due to their use of much longer intervals. They also investigate commercial benchmarks, and find that the results are not as accurate as the SPEC results.

Perelman et al. [120] look at cross-binary simulation points. These rely on code path traces instead of plain BBVs, which allows the SimPoint methodology to be applied across different compilations and even different architectures (as long as they are compiled from the same source code). This does require a more complicated data collection infrastructure, and requires gathering data on all architectures of interest. Using CMP$im [75] they find error similar to using regular SimPoints. In our work we generate cross-platform SimPoints by using a cross-platform simulator, which is a much simpler solution.

Nair and John's [108] work on SPEC CPU2006 SimPoints postdates ours.

They use PinPoint and performance counters on a Pentium 4, simulating up to 30 simulation points. They find average error of 2.45% for SPEC CPU2006 and 2.15% for SPEC CPU2000. This is better than our results, but they simulate more of the benchmarks by at least a factor of three.

Ganesan et al. [55] look at SimPoint results for SPEC CPU2006 using Alpha binaries on the sim-alpha simulator. No validation to real hardware is performed.

2.3 Performance Counter Validation

We notice irregularities when validating our BBV generation methods using hardware performance counters, leading us to validate the counters themselves. The previous work on the topic is not as comprehensive as our investigations, first presented at IISWC'08 [153].

Black et al. [24] use performance counters to investigate the total number of retired instructions and cycles on the PowerPC 604 platform. Unlike our work, they compare their results against a cycle-accurate simulator. The study uses a small number of benchmarks (including some from SPEC92), and the total number of instructions executed is many orders of magnitude fewer than in our work.

Patil et al. [115] validate SimPoint generation using CPI from Itanium performance counters. They compare different machines, but only the SimPoint-generated CPI values, not the raw performance counter results.

Sherwood et al. [132] compare results from performance counters on the Al-

pha architecture with SimpleScalar [13] and the Atom [135] DBI tool. They do not investigate changes in counts across more than one machine.

Korn, Teller, and Castillo [78] validate performance counters of the MIPS R12000 processor via microbenchmarks. They compare counter results to estimated (simulator-generated) results, but do not investigate the `instructions_graduated` metric (the MIPS equivalent of retired instructions). They report up to 25% error with the `instructions_decoded` counter on long-running benchmarks, though this is possibly due to the 20% error inherent in the simulator itself [43].

Maxwell et al. [95] look at accuracy of performance counters on a variety of architectures, including a Pentium III system. They report less than 1% error on the retired instruction metric, but only for microbenchmarks and only on one system.

Mathur and Cook [94] look at hand-instrumented versions of nine of the SPEC 2000 benchmarks on a Pentium III. They only report relative error of using sampled versus aggregate counts, and do not investigate overall error.

DeRose et al. [40] look at variation and error with performance counters on a Power3 system, but only for startup and shutdown costs. They do not report total benchmark behavior.

Zaparanuks et al. [173] investigate the accuracy of the cycle count on various x86 processors, as gathered by three different measurement infrastructures.

Mytkowicz et al. [105] investigate sources of non-deterministic execution, but look at causes for variations in run-time rather that retired instruction count.

Keeton et al. [76] use performance counters to thoroughly investigate the behavior or a parallel Pentium Pro based system. They swap CPU boards to vary cache sizes, allowing analysis to investigate changing hardware parameters. They did not compare against simulators.

2.4 Single-core DBI-Based Simulation

The DBI-based simulation methodology we use is inherently similar to trace-based simulation [144]. The idea of generating traces on the fly and feeding architectural simulation is not new. Our contribution is in validating the generated results against reduced input methods, hardware performance counters and cycle-accurate simulators.

2.4.1 Valgrind

Valgrind [113] is a dynamic binary instrumentation tool for the PowerPC, x86, x86_64 and ARM architectures. It is a generic and flexible DBI utility originally designed to detect application memory allocation errors. It comes with a single-core memory simulator called cachegrind.

2.4.2 Pin

Pin [87] is a fast DBI tool that runs on Intel architectures (including x86, x86_64, and Itanium), and supports the Linux and Windows operating systems. Pin

comes with some simple cache and branch simulator tools. It can be used to generate more complicated cache simulations, see CMP$im in Section 2.5.1.

2.4.3 Qemu

Qemu [18] is a DBI-based simulator that can simulate a large number of platforms, and also can simulate full operating systems. Qemu has no native cache simulation; any simulation done has to be patched into the binary. It is the only DBI tool that we investigate that can simulate cross-platform. We use a patched version of Qemu in conjunction with the Dinero [48] cache simulator in our WDDD'08 [152] work.

2.4.4 TAXI

Vlaovic and Davidson develop TAXI [148], which uses a Bochs-based front end to generate traces that are fed to a cycle-accurate simulator modeling an earlier x86 machine. They attempt to validate this method using performance counters, and find their major limiting factor to be lack of documentation for the architecture they are trying to model.

2.5 Multi-core Simulation

A number of academic cycle-accurate simulators have various levels of multi-core simulation support, the most popular being SESC [125], m5 [22], and Sim-

ics/GEMS [91]. We find these to be slow, and look to DBI for speed gains. Some simulators incorporate DBI methods for speed.

2.5.1 CMP$im

CMP$im [75, 74] is the most similar project to ours. The Pin DBI tool feeds the results from x86 simulation into a custom CMP cache simulator. Their results match an unspecified cycle-accurate model to within 13% (4% for benchmarks with low branch predictor misses) on SPEC CPU2006 with full input sets. They also run `ammp` from SPEC OMP and multi-programmed SPEC CPU2006 workloads (no validation was done on these results). Their cache-simulator implements a MESI-like coherence protocol. It is configurable in the number of levels, privacy, inclusion, associativity, allocation, and replacement policy. Unlike our work they only compare results against a simulator and not against actual hardware. Their simulation runs at a speed of 4-10MIPS.

2.5.2 Other

PTLSIM [172, 171], is a cycle-accurate simulator that uses DBI internally for speed. It is described more fully in Section 2.6.

Goldschmidt and Hennessy [57] investigate multi-threaded trace simulation, as compared to cycle-accurate simulation. They find the methods to be equivalent, except in cases where synchronization matters or where the metric measured has a small value.

Li et al. [84] generate traces using IBM's Turandot/PowerTimer (a cycle-accurate simulator) but then use these traces multiple times to feed Zauber, a cache simulator. Re-using the traces mitigates the overhead of using a cycle-accurate simulator.

Lee et al. [81] propose Composable Performance Regression which uses uniprocessor and contention models to predict multiprocessor performance. Once trained, the models can predict multiprocessor performance with median errors of under 7%.

Donald and Martonosi [47] create a parallel version of PowerPC SimpleScalar that can run CMP simulations, and is multi-threaded itself, giving a speedup of 2-3x on a multi-core system. This is still much slower than the benefits achievable by using DBI.

Muzahid et al. [103] detect data races using a Pin-based simulation method that feeds into an unspecified MESI cache simulator. They do not investigate performance or perform any validation.

Luo et al. [89] investigate speculative threads using a custom version of SimpleScalar fed by Pin. They do not validate their results or comment on performance.

2.6 Cycle-Accurate x86 Simulators

The x86 architecture has been the dominant desktop platform for a long time, and more recently it has begun dominating in server and high-performance computing situations. There is an ongoing push to use the architecture more

frequently in embedded systems as well. Any architectural study that avoids investigating x86 limits the relevance of the results. Unfortunately academic simulators are just now catching up to using x86 and many studies still use obsolete RISC architectures.

Loh et al. [86] present Zesto, which is a detailed x86 cycle accurate simulator based on SimpleScalar. It is designed with accuracy, not speed in mind. They have validated it versus wall-clock time on a series of microbenchmarks and found around 5% error.

m5 [22] has recently acquired x86 support (with many contributions by us). It has not been validated except by the work in this thesis. It currently cannot run in detailed out-of-order or in-order modes.

PTLsim [172, 171], is a DBI-based full-system simulator that runs x86 binaries. There is an SMT mode available but it uses a simplistic cache coherence scheme and does not model system memory at all. MPTLsim [174], an enhanced CMP version, is described but is not currently available.

2.7 Simulator Validations

Cycle-accurate simulators are often used without concern that results match real hardware. This limits architectural studies, as the magnitude of error in the results is unknown. We list previous studies that attempt validation of simulators.

Gibson et al. [56] validate various MIPS simulators against their R10000-based FLASH system. They find that even their most carefully designed simulators have surprisingly large errors. They, like we, call into question the value

of highly detailed simulators that are not validated against real hardware.

Black et al. [24, 25] create a model of the PowerPC 604 processor and validate it using hardware performance counters. They use a small set of benchmarks for validation, and try to reduce error. Interestingly, they find that fixing bugs in the simulator can actually increase the error in simulation because previous errors masked other bugs.

Desikan, Burger, Keckler and Austin [42, 43] validate the sim-alpha cycle-accurate simulator. They find that the generic sim-outorder simulator has upwards of 40% error, and even a fine-tuned attempt to match an actual Alpha machine still yields errors of around 15%. They run 22 of the SPEC CPU2000 benchmarks.

SimOS [127] is a full-system simulator. It was the first simulator to use DBI internally; its DBI implementation is called Embra [157]. Is it only 3-9x slower than actual hardware, although with the cache simulator enabled it is 7-20 times slower. It models a 32-bit MIPS R3000 and can run SGI IRIX. Parallel Embra can run parallel simulations which scale with multiple host cores. It models DASH-like directory memory coherence and has been validated against the MIPSy simulator (and MIPSy has been validated against a real machine, within 1-2% for uniprocessor) [128, 17]. Currently the project is not under development, although a version for PowerPC running AIX is developed by IBM [2] and a version that can run Linux is also developed [159].

XTREM [37] is a validated ARM simulator. It matches real hardware to within 4% for thermal measurements and 7% on IPC using some MiBench and Java benchmarks. The IPC numbers are collected using hardware performance

counters. XEEMU [66] is another ARM simulator validated with performance counters. They claim better results than XTREM. Varma et al. [147] look at power estimation on ARM using a simulator based on Intel's Xsim with is a simulator found to be with 2% for hardware memory accesses.

SIGMA: [41] is a memory system simulator validated to match real hardware within 1% using performance counters for a Power3 system. They only validate against one benchmark, `swim`, from SPEC CPU2000.

Barroso et al. [16] use hardware counters to validate SimOS on Alpha, as well as to characterize a memory subsystem. They also use the static binary instrumentation tool ATOM, but they in the end do not elaborate on their use of ATOM to gather traces.

2.8 Multi-processor Phase Detection

Many of the reduced execution methods previously mentioned, including SimPoint, will not work for multi-processor workloads. This severely limits multi-processor studies. Some attempts have been made to solve this problem.

Perelman et. al [121] find multi-processor SimPoints by gathering info for each thread individually, then aggregating the count.

Namkung et al. [109] synthesize samples from similar phase combinations. They find that they can reduce sampling by 90% with error of less than 5%.

Van Biesbrouck et al. [146] use a technique called a Co-Phase Matrix to combine single-threaded phase behaviors in order to estimate performance on SMT

systems. They found an error rate of 4% while only requiring 1% to be run for 28 pairs of Alpha SPEC CPU2000 benchmarks on the m5 simulator. This work is extended [145] to consider multiple benchmark starting points for higher accuracy.

Ekman and Stenstrom [50] use matched-pair comparison in conjunction with statistical sampling to reduce the amount of simulation needed for multiprocessor simulations.

Gonzalez et al. [58] propose using hardware performance counters in conjunction with density-based clustering algorithms to detect phases in parallel applications.

2.9 Deterministic Execution

Comparing performance results of CMP systems and simulators is difficult, as inherent non-determinism in the executions make it nearly impossible to compare results fairly. There have been many attempts at creating practical deterministic execution environments for CMPs; the methods proposed often require hardware modification and thus are not available on commodity processors. Examples of this are See Capo [98], DMP [44], Delorean [97], and Flight Data Recorder [165].

The most promising implementation that requires no hardware modification is Kendo [114]. They use performance counters to enforce deterministic context switching. The retired_stores counter is used; they, like us, found that on x86 the retired_instructions counter includes interrupt counts. They modify

the pthreads package to have a new type of deterministic lock. When running in deterministic mode there is an overall overhead of 16%.

Pereira et al. [117] present a method of deterministic execution that uses DBI instrumentation to gather dependence information. Data collection is 27x slower, but running in a simulator is actually slightly faster due to elimination of stalls.

Alameldeen et al. [8] propose using random perturbations and statistical methodology to mitigate non-determinism in simulation.

Narayanasamy et al. [110] log operating system effects in order to have deterministic multi-threaded workloads, but only when simulating multiple threads on a single core.

Lepak et al. [83] enhance a simulator to enable deterministic execution by recording various sources of non-determinism.

2.10 Performance Counter based CPI Prediction

DBI simulations have no concept of cycle time, making prediction of methods such as CPI or IPC difficult. Various other groups have looked at estimating cycles from other performance metrics.

Amato et al. [9] use performance counters on the R10000 to predict parallel application performance.

Marin et al. [90] predict execution time and cache misses on R12000 processors, and compare the results with hardware performance counters.

Luo et al. [88] estimate CPI values based on memory performance counter results on MIPS R10000. They find good results.

Eyerman et al. [52] use interval analysis on out of order processors and try to determine the causes of stalls that impact CPU. The Power5 processor has performance counter hardware dedicated to generating these CPI stacks.

Bhargava et al. [21] enhance CPI numbers by modeling speculative instruction execution when generating program traces. A "resurrection" tree can be used to simulate wrong-path execution even without having the original binary available.

CHAPTER 3

METHODS OF REDUCING SIMULATION TIME

A common way of reducing simulation time is using reduced execution methods. This involves running only a small part of a workload and extrapolating total behavior. This inherently adds error to the results, but the dramatically decreased runtime is often deemed worth it.

Running reduced inputs can have problems besides accuracy. For one, not running full inputs means the final benchmark results are not generated, which is an important step in determining if the simulator is working properly. Subtle bugs that are not enough to crash simulation but different enough to skew results can be hidden if the program subset being run does not generate I/O that can be compared to known good results. Another problem with reduced inputs is the loss of results that can only be observed over relatively long time periods. For example, temperature fluctuations happen on the order of many seconds, and reduced input methods often reduce simulation times to sub-second lengths of time.

Yi et al. [168] investigate common methods of speeding up simulations, which they break up into six categories (some additional methods are described in Section 2.1):

- Representative sampling (SimPoint [132]),
- Statistics based sampling (SMARTS [163]),
- Reduced input sets (such as training inputs, or MinneSPEC [77]),
- Simulating the first X Million instructions,
- Fast-forwarding Y Million instructions and simulating X Million, and

- Fast-forwarding Y Million, performing architectural warmup, then simulating X Million.

They conclude that SimPoint and SMARTS give the most accurate results, with differences in the 10% range. The other methods can have upward of 50% difference when compared to running full benchmarks. They investigated 10 years (from 1995 to 2005) of HPCA, ISCA, and MICRO papers and found that over 70% use reduced simulation methods. This shows how critical fast simulation is to the architectural community, and how important it is to understand the accuracy tradeoffs introduced by these methods.

We evaluate various of the reduced execution methods in order to compare the results with our dynamic binary instrumentation based approach that uses full input sets.

3.1 Running a Small Portion from the Beginning

The simplest form of reduced execution is simply to start at the beginning and execute for some number of instructions, usually a few billion. It turns out that this has poor accuracy, as often the beginning of a program is one-time initialization and startup routines and is not representative of full program execution. We look at this method in our analysis.

3.2 Un-guided Fast-forwarding

Another method is to fast-forward deeper into a program (most simulators support running a faster, functional, mode that can then be switched into slower cycles-accurate mode). Usually the program is fast-forwarded by a billion or more instructions before starting detailed simulation. This usually avoids the startup region of a program, but it is still not necessarily representative of the rest of the program. We also look at this method in our analysis.

3.3 Reduced Input Sets

Yi et al. found that using reduced input sets, such as MinneSPEC or the SPEC training input sets, had worse accuracy than SimPoint while requiring much more execution. We do present some results for the SPEC training inputs in Section 3.5.3 which agree with that analysis.

3.4 Statistics-based Sampling

Statistics based sampling, such as SMARTS [163]), is a method of reducing runtime by gathering detailed statistics from various parts of the execution. It has high-accuracy, but it requires large amounts (gigabytes) of disk space. Yi et al. found that the results were not much better than using multiple SimPoints. We did not investigate this type of reduced execution.

3.5 SimPoint

SimPoint [62, 118, 119, 131, 132] exploits the phase behavior of programs. Many applications exhibit cyclic behavior: code executing at one point in time behaves similarly to code running at some other point. Entire program behavior can be approximated by modeling only a representative set of intervals (in our case, *simulation points* or SimPoints).

Figures 3.1, 3.2, and 3.3 show examples of program phase behavior at a granularity of 100M instructions; these are captured using hardware performance counters from representative SPEC CPU2000 benchmarks. Each figure shows two metrics: the top is L1 D-Cache miss rate, and the bottom is cycles per instruction (CPI). Figure 3.1 shows `twolf`, which exhibits almost completely uniform behavior. For this type of program, one interval is enough to approximate whole-program behavior. Figure 3.2 shows the `mcf` benchmark, which has more complex behavior. Periodic behavior is evident: representative intervals from the various phases can be used to approximate total behavior. The last example, Figure 3.3, shows the extremely complex behavior of `gcc` running the `200.i` input set. Few patterns are apparent; this type of program is difficult to approximate with the SimPoint methodology (smaller phase intervals are needed to recognize patterns, and variable-size phases are possible, but choosing appropriate interval lengths is non-trivial). A complete set of CPI phase plots for x86 and x86_64 can be found in Appendix E.

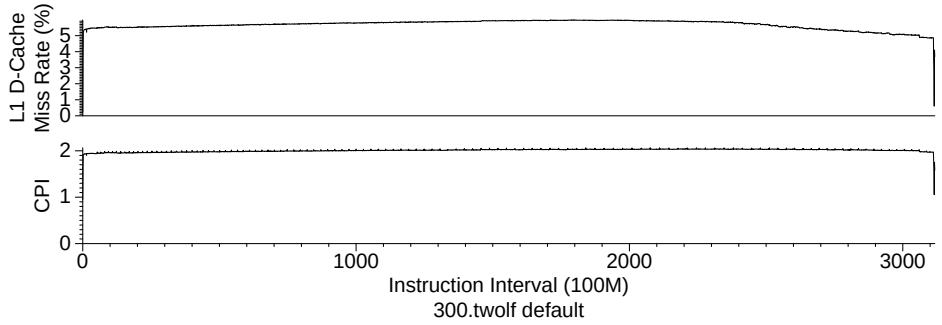

Figure 3.1: L1 Data Cache and CPI behavior for `twolf`: behavior is uniform, with one phase representing the entire program.

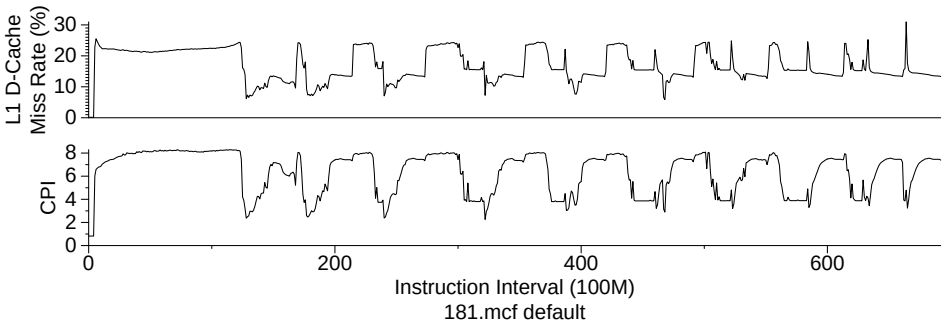

Figure 3.2: L1 Data Cache and CPI behavior for `mcf`: several recurring phases are evident.

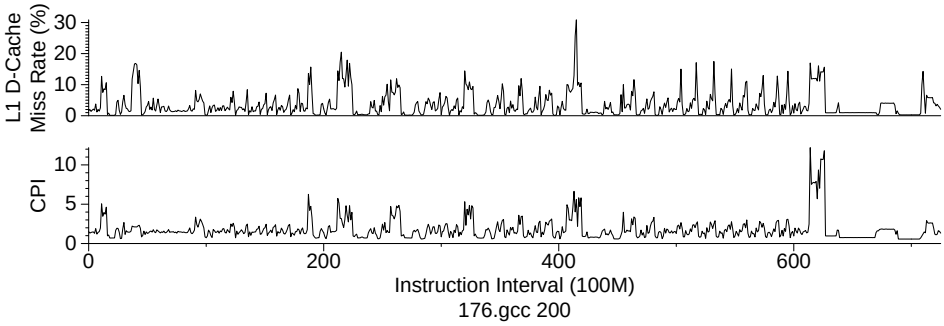

Figure 3.3: L1 Data Cache and CPI behavior for `gcc.200`: this program exhibits complex behavior that is hard to capture with phase detection.

3.5.1 BBV Generation

To generate the simulation points for a program, the SimPoint tool needs a Basic Block Vector (BBV) describing the code's execution. Dynamic execution is split into intervals (often fixed size, although that is not strictly necessary). Interval size is measured by number of committed instructions, usually 1M-300M instructions. Smaller sizes enable finer grained phase detection; larger sizes mitigate warmup error when fast-forwarding (without explicit state warmup) in a simulator. We use 100M instruction intervals, which is a common compromise. During execution, entry into all basic blocks is tracked along with a count of how many times each block is executed. The block count is weighted by the number of instructions in each block to ensure that instructions in smaller basic blocks are not given disproportionate significance. When total instruction count reaches the interval size, the basic block list and frequency count are appended to the BBV file.

The SimPoint methodology uses K-means clustering of the BBV file to find simulation points of interest. The algorithm selects one representative interval from each phase identified by clustering. The number of phases can be specified directly, or the tool can search within a given range for an appropriate number of phases.

The final step in using SimPoint is to gather statistics for all chosen simulation points. For multiple simulation points, the SimPoint tool generates weights to apply to the intervals. By scaling the statistics by the corresponding weights, an accurate approximation of entire program behavior can be estimated quickly (within a small fraction of whole-application simulation time).

The SimPoint website only provides BBV generation tools using ATOM [135] and SimpleScalar sim-alpha [13]. These are useful for experiments involving the Alpha processor, but that architecture has declined in significance. There is a PinPoints tool that enabled generation of BBVs using Intel's Pin [87] tool, but that only works for Intel supported architectures. We investigate using other tools to generate BBVs for a wider range of architectures.

We modify the Qemu [18] and Valgrind [113] Dynamic Binary Instrumentation tools to generate SimPoint BBV files. The changes to Qemu are available from our website [3] and are also shown in Appendix J. The tool we develop for Valgrind, `exp-bbv`, was merged into the main Valgrind project as of versions 3.5 (the code is also included as Appendix I). We tried using DynInst [29] to generate BBV files, but we were unsuccessful. Unfortunately the version of the tool available at the time only worked with dynamically linked applications and had a large overhead, often exceeding 4GB of RAM used for some benchmarks.

To evaluate our BBV generation methods, we compare results gathered on the x86 architecture, as this is the one architecture supported by Qemu, Valgrind and Pin. Figure 3.4 shows architectures supported by each tool.

3.5.2 x86 Evaluation

To evaluate the BBV generation tools, we use the SPEC CPU2000 [136] and CPU2006 [138] benchmarks with full reference inputs. We compile the benchmarks on SuSE Linux 10.2 with gcc 4.1 and `-O2` optimization (except for `vortex`, which we compile without optimization because it crashes, otherwise). We link binaries statically to avoid library differences on the machines we use

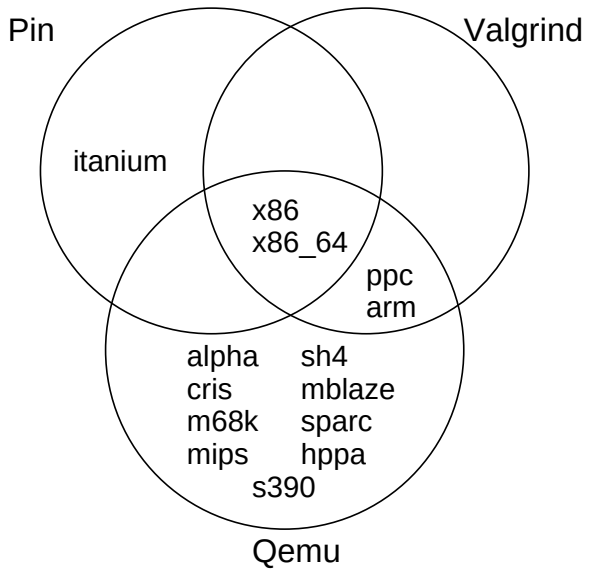

Figure 3.4: Architectures supported by Pin, Qemu, and Valgrind: x86 is the ideal platform for comparison, as it is well supported by all three of the tools.

to gather data. The choice to use static linking is not due to tool dependencies; all three handle both dynamic and static executables. We use the Perfmon2 [51] interface to gather hardware performance counter results for the platforms described in Table 3.1.

We use the Cycles Per Instruction (CPI) metric to evaluate our tools. The performance counter infrastructure is set to dump the cycles performance counter results every 100M instructions. The same performance counter data are used to evaluate all three tools, to avoid any variation between runs. Basic Block Vector files are generated using the three tools, and SimPoint version 3.2 is used to generate the simulation points and weights. We calculate actual overall CPI for the benchmarks by using the performance counter data, and use this as a basis for our error calculations. Note that calculated statistics are ideal, with full warmup. If we were analyzing via a simulation, the results would likely

Table 3.1: Machines used for x86 SimPoint evaluation.

type	frequency	memory	L1 I/D	L2/L3 Cache	performance counters used
Pentium Pro	200MHz	256MB	8KB/8KB	512KB	inst_retired, cpu_clk_unhalted
Pentium II	400MHz	256MB	16KB/16KB	512KB	inst_retired, cpu_clk_unhalted
Pentium III	550MHz	512MB	16KB/16KB	512KB	inst_retired, cpu_clk_unhalted
Itanium	800MHz	1GB	16KB/16KB	96KB/3MB	ia32_inst_retired, cpu_cycles
Atom N270	1.6GHz	1GB	32KB/24KB	512KB	instructions_retired, unhalted_core_cycles
Core Duo	1.66GHz	1GB	32KB/32KB	1MB	instructions_retired, unhalted_core_cycles
Athlon MP	1.733MHz	512MB	64KB/64KB	256KB	retired_instructions, cpu_clk_unhalted
Athlon64 X2	2GHz	1GB	64KB/64KB	512KB	retired_instructions, cpu_clk_unhalted
AMD Phenom	2.2GHz	2GB	64KB/64KB	512MB/2MB	retired_instuctions, cpu_clk_unhalted
Core2 Q6600	2.4GHz	2GB	32KB/32KB	4MB	instructions_retired, unhalted_core_cycles
Pentium 4	2.8GHz	2GB	12Kμ/16KB	512KB	instr_retired:nbogusntag, global_power_events:running
Pentium D	3.46GHz	4GB	12Kμ/16KB	2MB	instr_retired:nbogusntag, global_power_events:running

vary in accuracy depending on how architectural state is warmed up after fast-forwarding between simulation points.

Figure 3.5 shows results for reduced input methods for the SPEC CPU2000 benchmarks across 12 different implementations of the x86 architecture. The results shown are the average error for CPI when compared against a full reference input run, as measured with hardware performance counters. Each machine has three sets of plots; one for detailed simulation of 100 million instructions, one for 500 million, and one for 1 billion.

The first plot in each set is just starting from the beginning of the program and simulating for some instructions. The results are universally bad, although

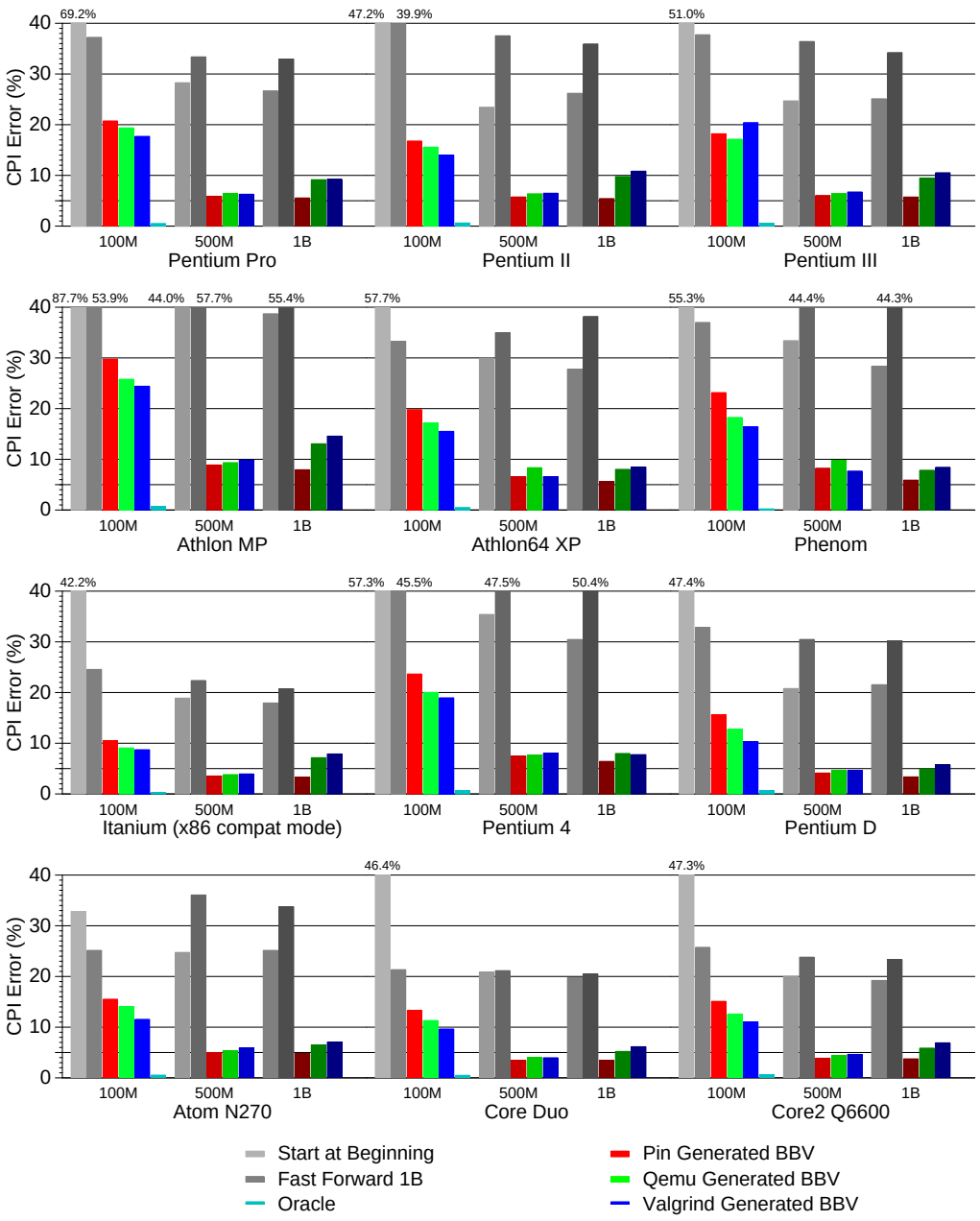

Figure 3.5: Average CPI error for SPEC CPU2000 when using first, unguided fast-forward, and SimPoint selected intervals on various x86 machines.

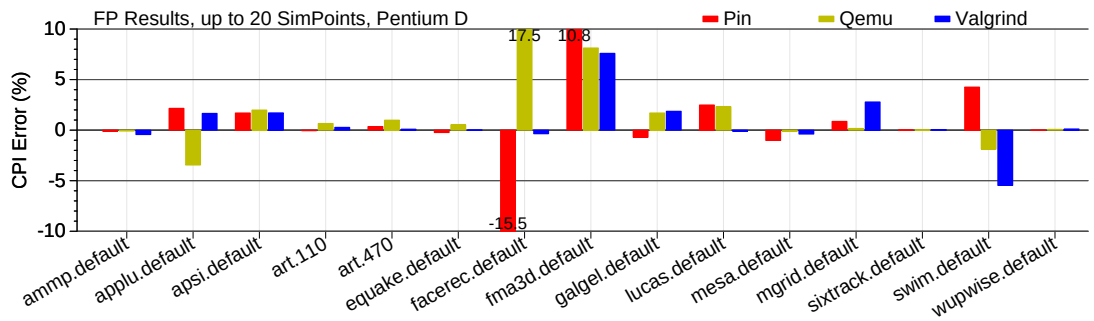

Figure 3.6: Percent error in CPI on a Pentium D when using up to 20 SimPoints on CPU2000 FP: the error with `facerec` and `fma3d` is due to extreme swings in the phase behavior that SimPoint has trouble capturing.

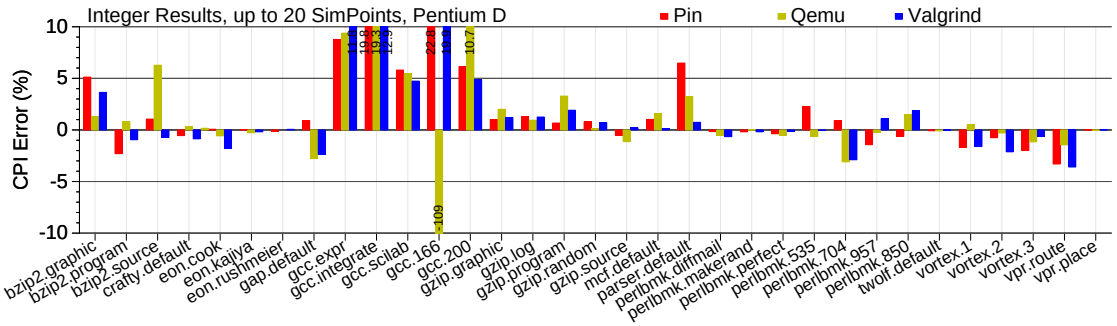

Figure 3.7: Percent error in CPI on a Pentium D when using up to 20 SimPoints on CPU2000 INT: the large error with the `gcc` benchmarks is due to spikes in the phase behavior that SimPoint does not capture well.

simulating more instructions can sometimes get results as close as 20% error.

The second plot in each set is fast forwarding by 1 billion instructions, in an attempt to avoid startup effects. This often, but not always, has better accuracy than starting from the beginning, and can also obtain results approaching 20% error, though usually higher.

The next three plots show the results using the SimPoint methodology, with

BBV files generated by Pin, Qemu and Valgrind respectively. Even when only simulating one simulation point, the results are much better than the unguided results. In general they are within the 10-20% error range. Moving on to up to 5 chosen SimPoints (approximately 500M instructions per benchmark) helps even more, with error in the 5-10% range for all machines. Moving on to simulating up to 10 SimPoints does not help much, and in fact it can have worse results! This might be unexpected, but there is no guarantee in the SimPoint methodology that adding more points helps error (Figure 3.19 shows this with regard to the x86_64 architecture).

The last plot shown is the "oracle" result shown. This shows how low the error would be if the optimal interval was picked for each benchmark. This is an extremely low value, which shows that each benchmark has an interval that matches program behavior well. Unfortunately this interval varies from machine to machine, so it would not be possible to have a tool that can find this in a generic fashion.

The thing to note about these results is the small amount of simulation time required. When allowing SimPoint to choose up to 10 simulation points per benchmark, the average error across all machines for CPI is roughly 5-10% for all machines tested while having a small amount of execution. The intervals chosen are not many; Pin chooses 354 SimPoints, Qemu 363, and Valgrind 346; this represents only 0.4% of the total execution length, making the simulations finish 250 times faster than if run to completion. It is reassuring that all three BBV methods pick a similar number of intervals, and in many cases they pick the same intervals.

Figures 3.6 and 3.7 break out the Pentium D results by benchmark. For float-

ing point applications, `facerec` and `fma3d` have significantly more error than the others. This is because those programs feature phases which exhibit extreme shifts in CPI from interval to interval, a behavior that SimPoint often has trouble capturing. The integer benchmarks have the biggest source of error, which is the `gcc` benchmarks. The reason `gcc` behaves so poorly is that there are intervals during its execution where the CPI and other metrics spike. These huge spikes do not repeat, and only happen for one interval; because of this, SimPoint does not weight them as being important, and they therefore are omitted from the chosen simulation points. These high peaks are what cause the actual average results to be much higher than what is predicted by SimPoint. It might be possible to work around this problem by choosing a smaller interval size, which would break the problematic intervals into multiple smaller ones that would be more easily seen by SimPoint.

We also use our BBV tools on the SPEC CPU2006 benchmarks. These runs use the same tools as for CPU2000, without any modifications. These tools yield good results without requiring any special knowledge of the newer benchmarks. We do not have results for the `zeusmp` benchmark for Valgrind; it uses a 1GB data segment which Valgrind was unable to handle. Unlike the CPU2000 results, we only have performance counter data from six of the machines. Many of the CPU2006 benchmarks have working sets of over 1GB, and many of our machines have less RAM than that. On those machines the benchmarks take months to run, with the operating system paging constantly to disk. The CPU2006 results shown in Figure 3.8 are as favorable as the CPU2000 results. When allowing SimPoint to choose up to 10 simulation points per benchmark, the average error for CPI is less than 10% for all of the BBV generation methods. Pin chooses 420 simulation points, Qemu 433, and Valgrind. This

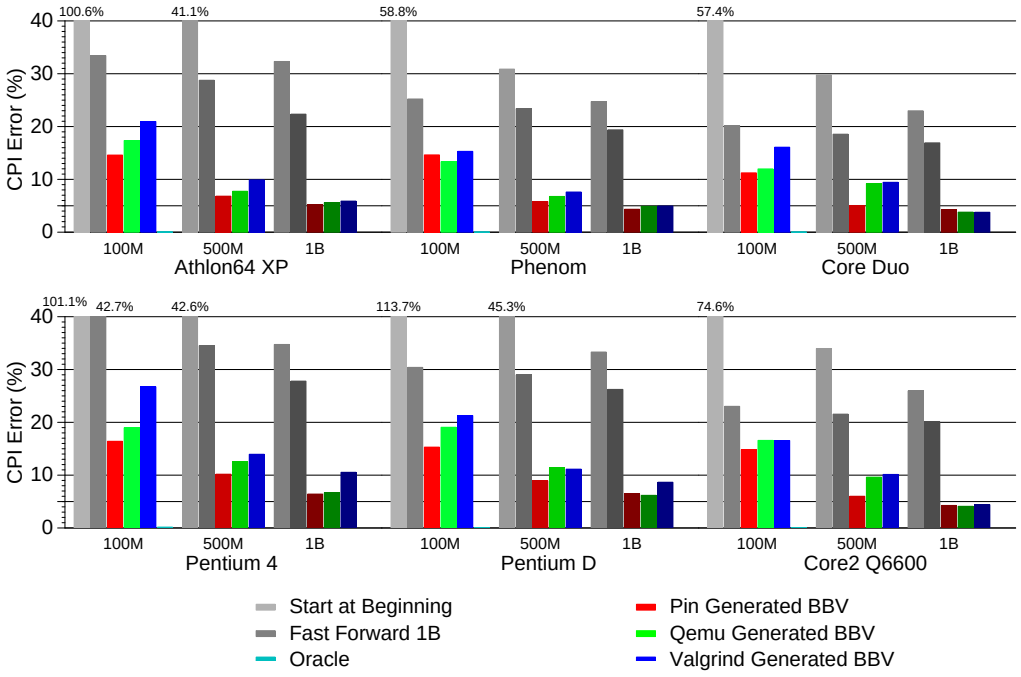

Figure 3.8: Average CPI error for CPU2006 on a selection of x86 machines when using first, unguided fast-forward, and SimPoint selected intervals.

would require simulating only 0.056% of the total benchmark suite. This is an impressive speedup, considering the long running time of these benchmarks.

Error when simulating the first 100M instructions peaks at over 100%, showing that this continues to be a poor way to choose simulation intervals. Fast-forwarding 1B instructions and then simulating produces average errors in the range of 20-40%. Using only a single simulation point again always does better than unguided simulation.

Figures 3.9 and 3.10 show CPI errors for individual benchmarks on the Pentium D machine. For floating point applications, there are outlying results for `cactusADM` and `GemsFDTD`. As with the CPU2000 results, the biggest source of error is from `gcc` in the integer benchmarks. The reasons are the same as

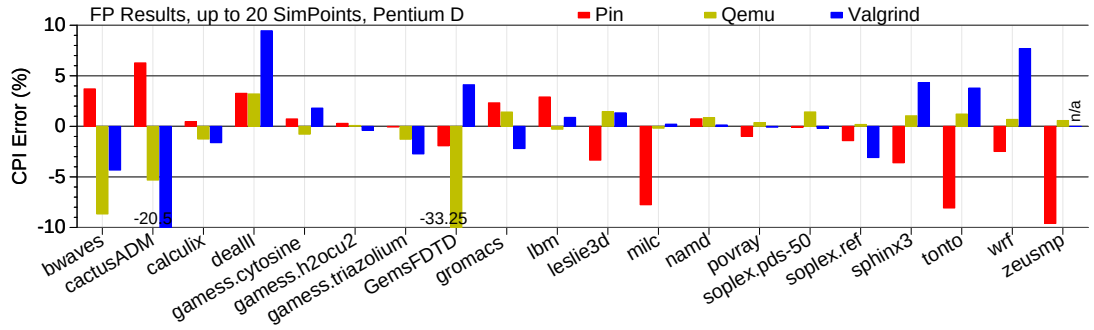

Figure 3.9: Percent error in CPI on a Pentium D when using up to 20 SimPoints on CPU2006 FP: the large variation in results for `cactusADM` and `GemsFDTD` are due to unresolved inaccuracies in the way the tools count instructions.

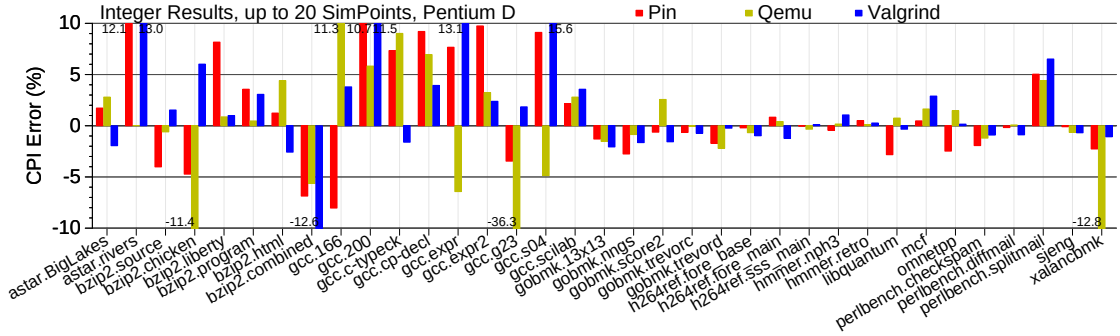

Figure 3.10: Percent error in CPI on a Pentium D when using up to 20 SimPoints on CPU2006 INT: the large error with the `gcc` and `bzip2` benchmarks is due to spikes in the phase behavior not captured by SimPoint.

described previously: SimPoint cannot handle the spikes in the phase behavior. The `bzip2` benchmarks in CPU2006 exhibit the same problem that `gcc` has. Inputs used in CPU2006 have spiky behavior that the CPU2000 inputs do not. The other outliers, `perlbench` and `astar` require further investigation.

Table 3.2: Machines used for x86_64 SimPoint evaluation.

Processor	Cores	Speed	Memory	L1 I/D Cache	L2/L3 Cache	Retired Instruction Counter Cycles Counter
AMD Phenom	4	2.2GHz	2GB	64KB/64KB	512MB/2MB	retired_instuctions, cpu_clk_unhalted
Core2 Q6600	4	2.4GHz	2GB	32KB/32KB	4MB	instructions_retired, unhalted_core_cycles
Pentium D	2x2	3.46GHz	4GB	12Kμ/16KB	2MB	instr_completed:nbogus, global_power_events:running

3.5.3 x86_64 Results

The x86_64 architecture is a 64-bit extension of the x86 architecture. While it is very similar to the x86 architecture, it has features that change program behavior. The move to 64-bits causes memory access widths to change, there are more registers (which reduces register spills), and by default SSE vector instructions can be used (this allows for saner floating point math and optimized memory transfers). We extend our original x86 SimPoint work by generating results for x86_64.

The machines used are described in Table 3.2. The SPEC CPU2000 benchmarks were used, compiled with `-O3 -msse3 -funroll-all-loops -ffast-math -static` using gcc-4.2. With that configuration, some of the `perlbmk` benchmarks and all of the `vortex` benchmarks fail to run due to memory access errors inherent in the benchmarks that are exhibited with recent compilers.

Unlike the x86 results, we use only SimPoints from Valgrind-generated BBV files. In Section 3.5.2 we show that the Valgrind generated BBV files have similar characteristics to those generated by other tools. We generate the BBV files using our `exp-bbv` tool as included in Valgrind 3.5.

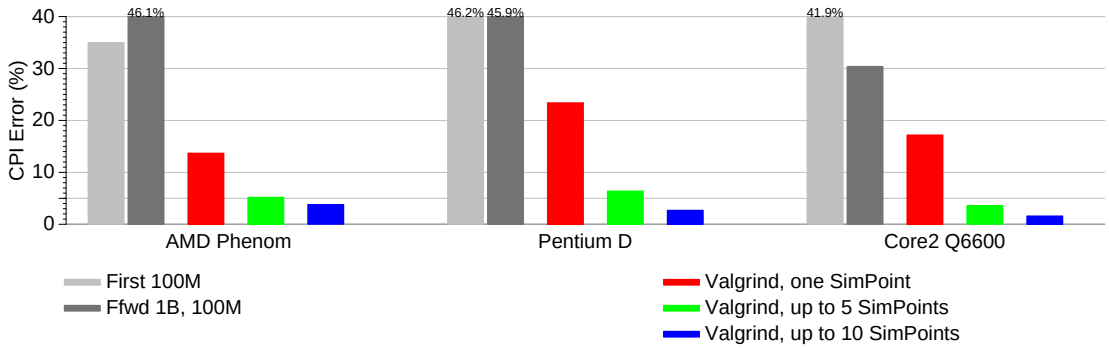

Figure 3.11: Average CPI error for CPU2000 on three x86_64 machines when using first, unguided fast-forward, and SimPoint selected intervals.

Figure 3.11 shows CPI error for three different x86_64 implementations on the SPEC CPU2000 benchmarks. On all of the machines, the SimPoint results are much better than the un-guided results. Increasing the number of simulation points helps accuracy, and on all machines accuracy of better than 5% can be found when using up to 10 SimPoints per benchmark. This is better than the average results found using the x86 binaries, even on the same machines. This is primarily due to the outliers being much better behaved on 64-bit systems, and since it is an average measure, it is the outliers which cause the high percent error results.

Figure 3.12 and 3.13 show broken out results for the Phenom when up to 10 SimPoints are used per benchmark. It is somewhat unsurprising to note that the outlying benchmarks are pretty much the same as the ones found for 32-bit x86 in Section 3.5.2.

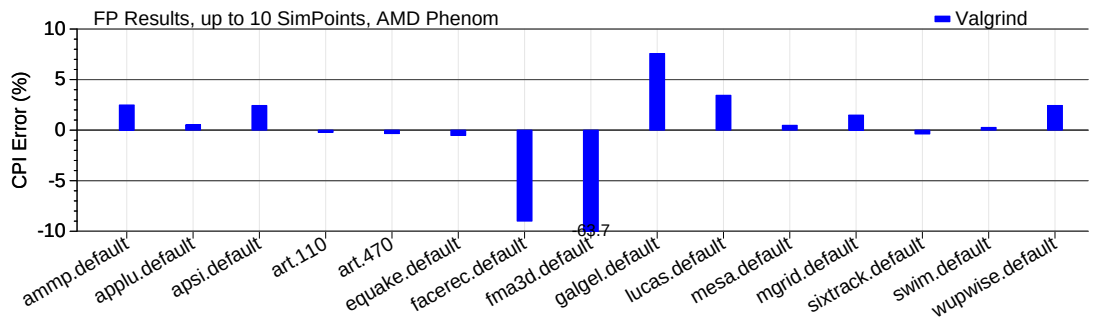

Figure 3.12: x86_64 CPI Error for SPEC CPU2000 floating point benchmarks

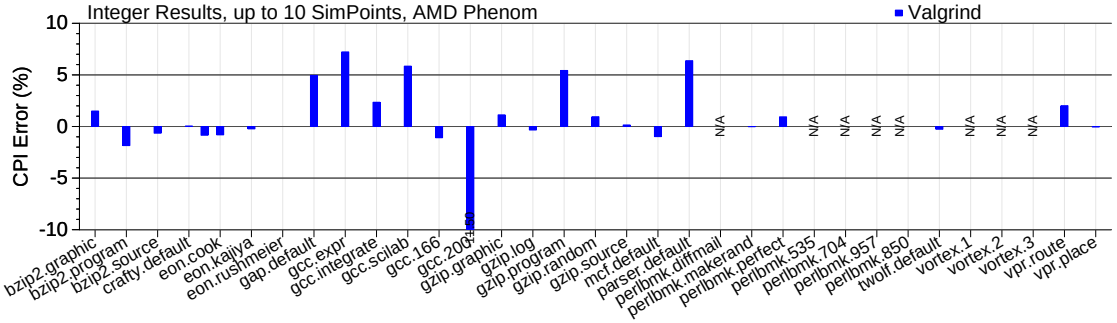

Figure 3.13: x86_64 CPI Error for SPEC CPU2000 integer benchmarks

3.5.4 Cross-Platform MIPS Results

A common situation found when performing architectural simulation is using simulators for machines for which you do not have any actual hardware. This makes for difficult development, involving setting up cross-compiler toolchains to generate binaries. It becomes hard to determine when bugs are in the toolchain or in the simulator when there is no real hardware for comparison. Generating SimPoints for an unavailable platform is also difficult; it might be tempting to just re-use SimPoints generated for another architecture, but this is not advisable. Figure 3.14 shows phase plots for the `mcf` benchmark across

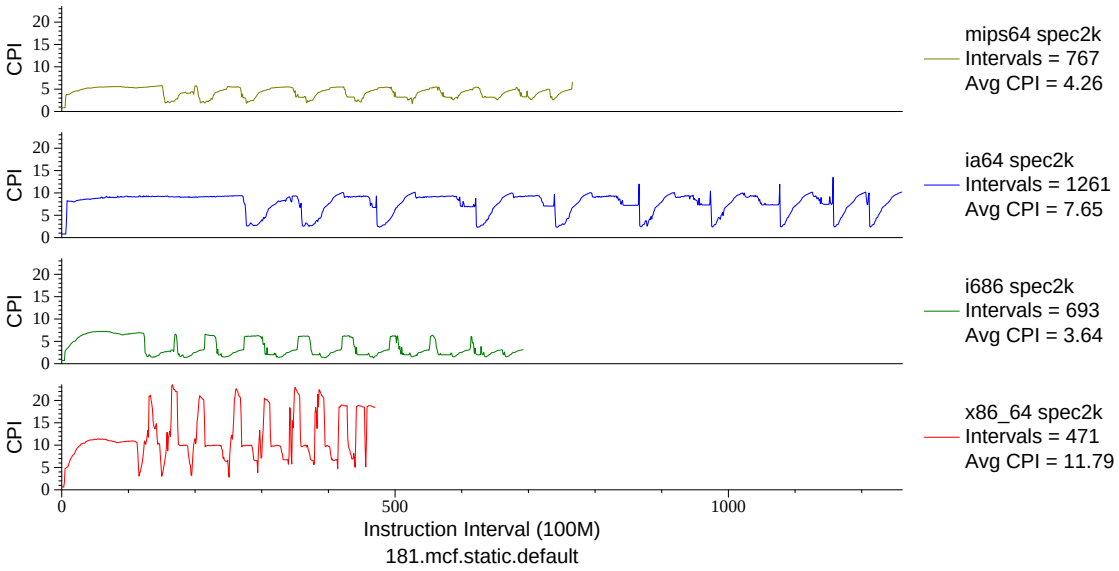

Figure 3.14: Phase plot for `mcf` across various architectures. While the phases look similar, the interval numbers are not.

multiple architectures. (A complete set of multi-architecture phase plots can be seen in Appendix F). While the phases are similar, the actual interval values are very different, and SimPoints generated for one of the architectures would not work for any of the others.

Another way to avoid generating SimPoints is to re-use those already generated by someone else. This can cause problems unless you have the exact same binaries used to generate the original SimPoints. Figure 3.15 shows that on x86 the compiler chosen and the compiler flags used can vastly affect the interval numbers for a benchmark (in this case, `equake`.

There have been studies done on the possibility of generating true cross-platform SimPoints [120], but the methods involve time-consuming profiling on multiple machines, and the results are not practical.

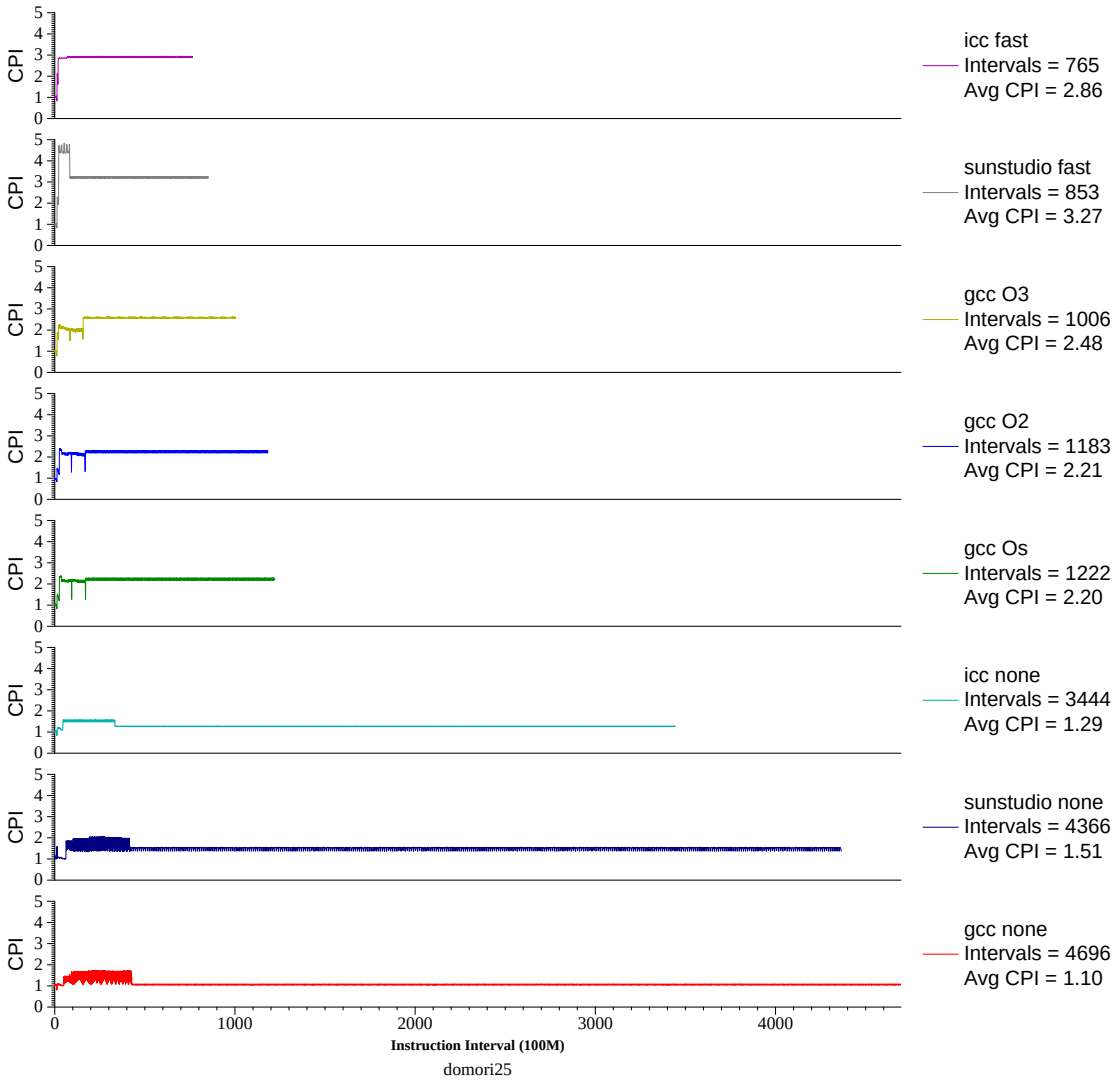

Figure 3.15: Phase plot for `equake` across various compilers are compile options. The interval numbers vary widely.

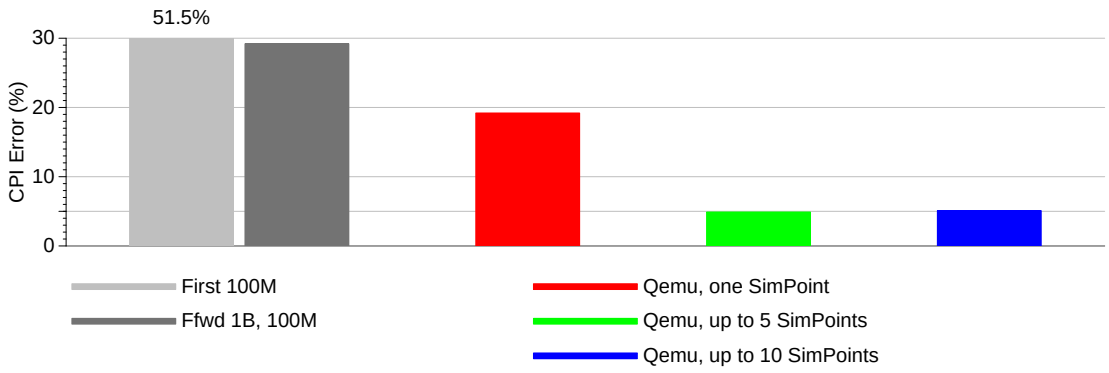

Figure 3.16: MIPS R12000 SimPoint results for SPEC CPU2000. The BBVs for the SimPoints were generated cross-platform on an x86 machine using Qemu

An option we explore is to use DBI simulation to generate BBV files for a different platform. The Qemu DBI tool can run executables cross-platform. By using our BBV-generation patched version of Qemu, we can generate BBV files for Alpha, SPARC, MIPS, PPC and ARM while still running on an x86 machine.

Figure 3.16 shows results using SimPoints generated for the MIPS architecture using MIPS binaries while running on an x86 machine. These SimPoints were then used on performance counter data collected on an actual MIPS R12000 processor. The results are very similar to those found for the other architectures investigated, and have 5% CPI error when using up to 5 SimPoints. This shows that Qemu is a valuable tool for generating SimPoints for platforms where native hardware is not available.

Figures 3.17 and 3.18 break out the results per-benchmark. The results are markedly different from the x86 and x86_64 results seen previously. The `gcc` benchmarks are not outliers, in this case `mcf` has a large error, and the floating point benchmarks are more of a problem.

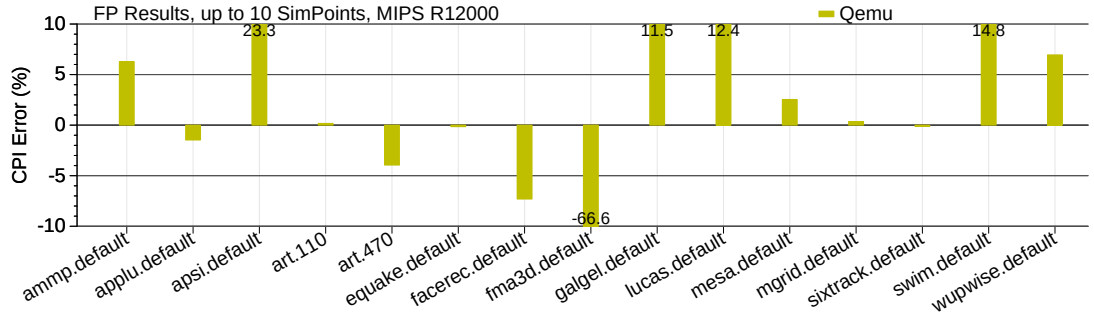

Figure 3.17: MIPS CPI Error for SPEC CPU2000 floating point

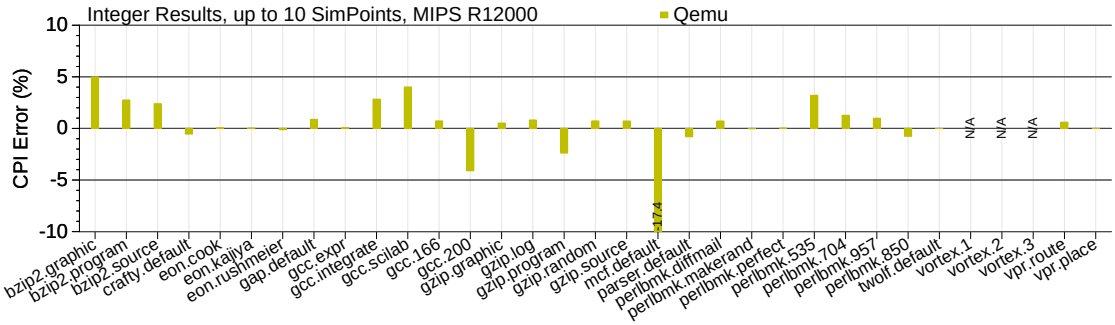

Figure 3.18: MIPS CPI Error for SPEC CPU2000 integer benchmarks

3.5.5 Summary

On actual x86 hardware, using the SimPoint methodology can give CPI error of under 10% while only running 0.4% of the total SPEC CPU2000 suite on full reference inputs across 12 different machines. Our code generates under 12% CPI error when running under 0.06% of SPEC CPU2006 (excepting `zeusmp`) with full reference inputs across 6 different machines.

We also investigate x86_64 and cross-platform generated MIPS SimPoints and find results that compare favorably to the x86 results.

3.6 SimPoint Limitations

While these results are good, there are some limitations to using this methodology. This error can only add to the error generated with cycle-accurate simulators (for example, 20% with sim-alpha [43]). Also, it is unclear if it is possible to use the SimPoint methodology for multi-threaded workloads (see discussion in Section 2.8.

We find more variation in our results than we originally expected. This led us to investigate our evaluation methods to try to determine the source of the differences. For example, we would expect that the different DBI tools, since they are running the same executables/inputs on the same machines with the same inputs, should have identical BBV files, but they do not. This turns out to be because the different DBI tools have different ideas of what constitute a basic block. For performance reasons the DBI tools try to have biggest blocks as possible, and will use "super-blocks" which unlike basic blocks can have multiple exits but only one entry. Also, the tools discover basic-blocks at run-time, so are often in the situation where a program will jump to a middle of a block (or on x86, it's even technically legal to jump to the middle of an *instruction*), which means that a new block has to be created out of the old one, and the DBI tools differ into how statistics are accounted in that situation. The SimPoint methodology generates different SimPoint files depending on the BBV inputs, and even a single extra instruction in a block can change which points are chosen. Because of this, even slight difference in BBV accounting can cause different results.

Even with the DBI differences, we found that even on real hardware the performance counts for retired instructions were different from machine to ma-

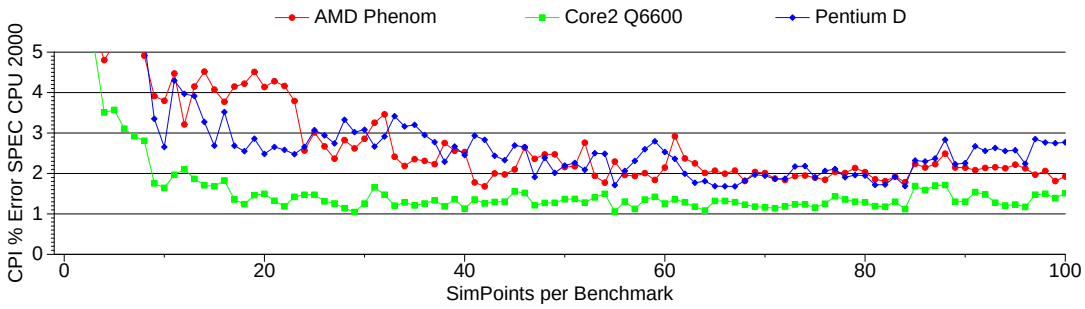

Figure 3.19: Percent average CPI error for SPEC CPU2000 as more SimPoints are added per benchmark. After 20 SimPoints the average does not decrease, even up to 100 points per benchmark (this is equivalent to running 2% of all of the benchmarks).

chine, which was often unexpected. To get accurate SimPoint results you need to have fairly accurate instruction counts, as you need to fast-forward to the exact start of the phase. On programs with a high amount of phase variability being a million instructions off could end up in a completely different phase than the one intended, causing poor results. This exposes many hardware counter and deterministic execution issues that we investigate it in detail in Chapter 4.

Another problem with the SimPoint methodology is that it is not possible to predict what the error will be. Figure 3.19 shows the average CPI error on three different x86_64 machines with SPEC CPU2000 as the number of SimPoints per benchmark is raised from 1 to 100. The error does not always decrease, and after a certain point (roughly around 20) a steady-state is reached and the error does not get better and in fact can get worse.

SimPoint is a valuable tool, and is much better than using unguided simulation. However, we believe that many of its limitations cannot be fully addressed, and thus suggest finding some way to run full input sets if at all possible.

CHAPTER 4

SINGLE-CORE VALIDATION CONCERNS

Hardware performance counters are a useful tool for validation. These counters are available on most modern processors, and keep track in real time of various architectural statistics. The counters must be used with caution, as hardware engineers are reluctant to certify the accuracy of the counters. Before using a counter in research, it needs to be checked to ensure it is delivering reasonable results.

When using hardware performance counters to validate the SimPoint methodology in Chapter 3 we noticed discrepancies in the results. Some of these could be attributed to variations in how the DBI tools generate BBV files, but some results indicate that the retired instructions counters were varying both run-to-run and across machines. These unexpected variations can be by as much as 2%. The retired instruction counter should not vary this much; it is high profile enough to be heavily debugged by hardware engineers. Retired instruction count is one of the few counters that should be the same for the same executable/input set across all implementations of an ISA.

In order to trust the results from our SimPoint study we investigate the accuracy of the retired instruction performance counter and how it relates to deterministic execution on the x86 architecture.

4.1 Hardware Performance Counters

When used in aggregate counting mode (as opposed to sampling mode), performance counters provide architectural statistics at full hardware speed with

minimal overhead. Most modern processors support some form of counters. Although originally implemented for debugging hardware designs during development, they have come to be used extensively for performance analysis and for validating tools and simulators. The types and numbers of events tracked and the methodologies for using these performance counters vary widely, not only across architectures, but also across systems sharing an ISA. For example, the Pentium III tracks 80 different events, measuring only two at a time, but the Pentium 4 tracks 48 different events, measuring up to 18 at a time. Chips manufactured by different companies have even more divergent counter architectures: for instance, AMD and Intel implementations have little in common, despite their supporting the same ISA. Verifying that measurements generate meaningful results across arrays of implementations is essential to using counters for research.

Comparison across diverse machines requires a common subset of equivalent counters. Many counters are unsuitable due to microarchitectural or timing differences. Furthermore, counters used for architectural comparisons must be available on all machines of interest. We choose a counter that meets these requirements: number of retired instructions. For a given statically linked binary, the retired instruction count *should* be the same on all machines implementing the same ISA, since the number of retired instructions excludes speculation and cache effects that complicate cross-machine correlation. When validating SimPoints (as described in Chapter 3) the retired instruction count was not as regular as expected. This count is especially relevant, since it is a component of both the Cycles per Instruction (CPI) and (conversely) Instructions per Cycle (IPC) metrics commonly used to describe machine performance.

The CPI and IPC metrics are important in computer architecture research; in the rare occasion that a simulator is actually validated [116, 37, 42, 152] these metrics are usually the ones used for comparison. Retired instruction count and IPC are also used for vertical profiling [64] and trace alignment [106], which are methods of synchronizing data from various trace streams for analysis.

Retired instruction counts are also important when generating basic block vectors (BBVs) for use with the SimPoint [62] tool. When investigating the use of DBI tools to generate BBVs [155], we find that even a single extra instruction counted in a basic block can change which simulation points the SimPoint tool chooses to be most representative of whole program execution.

All these uses of retired instruction counters assume that generated results are repeatable, relatively deterministic, and have minimal variation across machines with the same ISA. Here we explore whether these assumptions hold by comparing the hardware-based counts from a variety of machines, as well as comparing to counts generated by Dynamic Binary Instrumentation (DBI) tools.

4.1.1 Performance Counter Evaluation

We run experiments on multiple generations of x86 machines, listed in Table 4.1. All machines run the Linux 2.6.25.4 kernel patched to enable performance counter collection with the perfmon2 [51] infrastructure. We use the entire SPEC CPU2000 [136] and CPU2006 [138] benchmark suites with the full reference input sets. We compile the SPEC benchmarks on a SuSE Linux 10.1 system with version 4.1 of the gcc compiler and `-O2` optimization (except for `vortex`, which crashes when compiled with optimization). All benchmarks are statically linked

Table 4.1: Machines used for this study.

Processor	Speed	Bits	Memory	L1 I/D Cache	L2 Cache	Retired Instruction Counter / Cycles Counter
Pentium Pro	200MHz	32	256MB	8KB/8KB	512KB	inst_retired cpu_clk_unhalted
Pentium II	400MHz	32	256MB	16KB/16KB	512KB	inst_retired cpu_clk_unhalted
Pentium III	550MHz	32	512MB	16KB/16KB	512KB	inst_retired cpu_clk_unhalted
Pentium 4	2.8GHz	32	2GB	12Kμ/16KB	512KB	instr_retired:nbogusntag global_power_events:running
Pentium D	3.46GHz	64	4GB	12Kμ/16KB	2MB	instr_completed:nbogus global_power_events:running
Athlon XP	1.733GHz	32	768MB	64KB/64KB	256KB	retired_instructions cpu_clk_unhalted
AMD Phenom	2.2GHz	64	2GB	64KB/64KB	512KB	retired_instructions cpu_clk_unhalted
Core Duo	1.66GHz	32	1GB	32KB/32KB	1MB	instructions_retired unhalted_core_cycles
Core2 Q6600	2.4GHz	64	2GB	32KB/32KB	4MB	instructions_retired unhalted_core_cycles

to avoid variations due to the C library. We use the same 32-bit, statically linked binaries for all experiments on all machines.

We gather Pin [87] results using a simple instruction count utility via Pin version pin-2.0-10520-gcc.4.0.0-ia32-linux. We patch Valgrind [113] 3.3.0 and Qemu [18] 0.9.1 to generate retired instruction counts. We gather the DBI results on a cluster of Pentium D machines identical to that described in Figure 4.1. We configure `pfmon` [51] to gather complete aggregate retired instruction counts, without any sampling. The tool runs as a separate process, enabling counting in the OS; it requires no changes to the application of interest and induces minimal overhead during execution. We count user-level instructions specific to the benchmark.

We collect at least seven data points for every benchmark/input combination on each machine and with each DBI method. The CPU2006 benchmarks require at least 1GB of RAM to finish in a reasonable amount of time. Given

this, we do not run them on the Pentium Pro or Pentium II, and we do not run `bwaves`, `GemsFDTD`, `mcf`, or `zeusmp` on machines with small memories. Furthermore, we omit results for `zeusmp` with DBI tools, since they cannot handle the large 1GB data segment the application requires.

4.1.2 Sources of Hardware Counter Variation

We focus on two types of variation when gathering performance counter results. One is inter-machine variations, the differences between counts on two different systems. The other is intra-machine variations, those found when running the same benchmark multiple times on the same system. We investigate methods for reducing both types.

Specific Instructions Counted Differently

For instruction counts to match on two machines, the instructions involved must be counted the same way. If not, this can cause large divergences in total counts. On Pentium 4 systems, the `instr_retired:nbogusntag` performance counter counts `fldcw` as two retired instructions; on all other x86 implementations `fldcw` counts as one. This instruction is common in floating point code: it is used in converting between floating point and integer values. It alone accounts for a significant divergence in the `mesa` and `sphinx3` benchmarks. Table 4.2 demonstrates occurrences in the SPEC benchmarks where the count is over 100 million. We modify Valgrind to count the `fldcw` instructions, and use these counts to adjust results when presenting Pentium 4 data. It should be possible to use statistical methods to automatically determine which type of opcode

Table 4.2: Dynamic count of `fldcw` instructions, showing all benchmarks with over 100 million. This instruction is counted as two instructions on Pentium 4 machines but only as one instruction on all other implementations.

benchmark	fldcw instructions	% overcount
482.sphinx3	23,816,121,371	0.84%
177.mesa	6,894,849,997	2.44%
481.wrf	1,504,371,988	0.04%
453.povray	1,396,659,575	0.12%
456.hmmer retro	561,271,823	0.03%
175.vpr place	405,499,739	0.37%
300.twolf	379,247,681	0.12%
483.xalancbmk	358,907,611	0.03%
416.gamess cytosine	255,142,184	0.02%
435.gromacs	230,286,959	0.01%
252.eon kajiya	159,579,683	0.15%
252.eon cook	107,592,203	0.13%

causes divergence in cases like this; this is part of ongoing work. We isolated the `fldcw` problem by using a tedious binary search of the `mesa` source code.

Using the Proper Counter

Pentium 4 systems newer than model 6 support a `instr_completed:nbogus` counter, which is more accurate than the `instr_retired:nbogusntag` counter found on previous models. This newer counter does not suffer the `fldcw` problem described in Section 4.1.2. Unfortunately, all systems do not include this counter; our Pentium D can use it, but our older Pentium 4 systems cannot. This counter is not well documented, and thus it was not originally available within the `perfmon` infrastructure. We contributed counter support that has been merged into the main `perfmon` source tree.

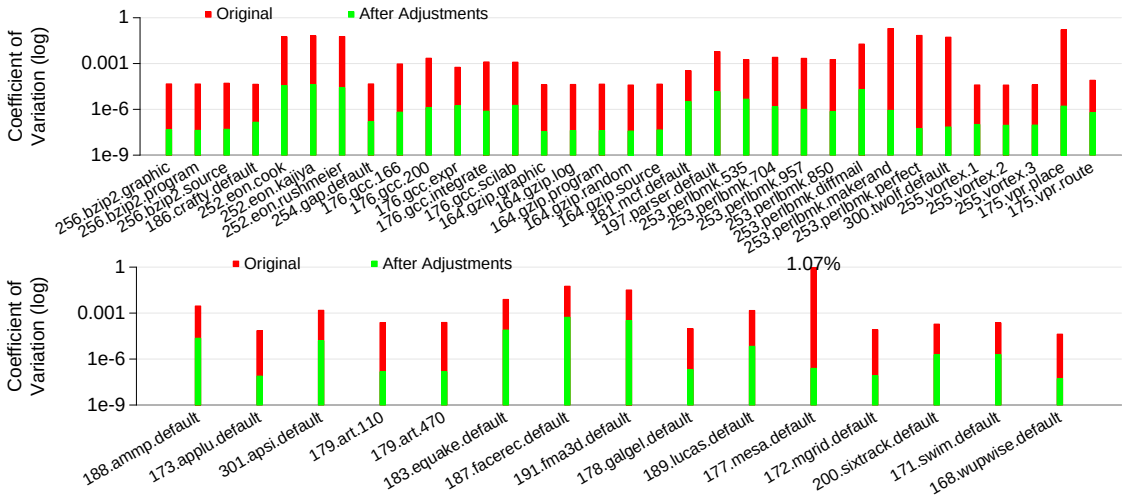

Figure 4.1: SPEC 2000 Coefficient of variation. The top graph shows integer benchmarks, the bottom, floating point. The error variation from `mesa`, `perlbmk`, `vpr`, `twolf` and `eon` are primarily due to the `fldcw` miscount on the Pentium 4 systems. Variation after our adjustments becomes negligible.

Processor Errata

There are built-in limitations to performance counter accuracy. Some are intended, and some are unintentional by-products of the processor design. Our results for our 32-bit Athlon exhibit some unexplained divergences, leading us to investigate existing errata for this processor [6]. The errata mention various counter limitations that can result in incorrect total instruction counts. Researchers must use caution when gathering counts on such machines.

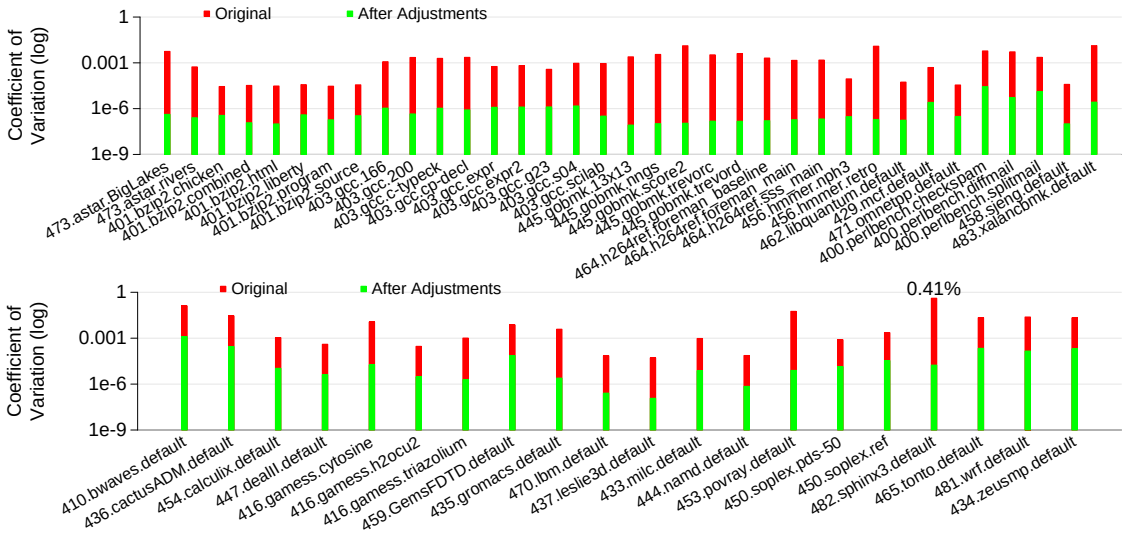

Figure 4.2: SPEC 2006 Coefficient of variation. The top graph shows integer benchmarks, bottom, floating point. The original variation is small compared to the large numbers of instructions in these benchmarks. The largest variation is in `sphinx3`, due to `fldcw` instruction issues. Variation after our adjustments becomes orders of magnitude smaller.

4.1.3 Counter Variation Findings

Figure 4.1 shows the coefficient of variation for SPEC CPU2000 benchmarks before and after our adjustments. Large variations in `mesa`, `perlbmk`, `vpr`, `twolf`, and `eon` are due to the Pentium 4 `fldcw` problem described in Section 4.1.2. Once adjustments are applied, variation drops below 0.0006% in all cases. Figure 4.2 shows similar results for SPEC CPU2006 benchmarks. Larger variations for `sphinx3` and `povray` are again due to the `fldcw` instruction. Once adjustments are made, variations drop below 0.002%. Overall, the CPU2006 variations are much lower than for CPU2000; the higher absolute differences are counterbalanced by the much larger numbers of total retired instructions. These results can be misleading: a billion-instruction differ-

ence appears small in percentage terms when part of a three trillion instruction program, but in absolute terms it is large. When attempting to capture phase behavior accurately using SimPoint with an interval size of 100 million instructions, a phase's being offset by one billion instructions can alter final results.

4.1.4 Intra-machine results

Figure 4.3 shows the standard deviations of results across the CPU2000 and CPU2006 benchmarks for each machine and DBI method. DBI results are shown, but not incorporated into standard deviations. In all but one case the standard deviation improves, often by at least an order of magnitude. For CPU2000 benchmarks, `perlbmk` has large variation for every generation method. We are still investigating the cause. In addition, the Pin DBI tool has a large outlier with the `parser` benchmark, most likely due to issues with consistent heap locations. Improvements for CPU2006 benchmarks are less dramatic, with large standard deviations due to high outlying results. On AMD machines, `perlbench` has larger variation than on other machines, for unknown reasons. The `povray` benchmark is an outlier on all machines (and on the DBI tools); this requires further investigation. The Valgrind DBI tool actually has worse standard deviations after our methods are applied due to a large increase in variation with the `perlbench` benchmarks. For the CPU2006 benchmarks, similar platforms have similar outliers: the two AMD machines share outliers, as do the two Pentium 4 machines.

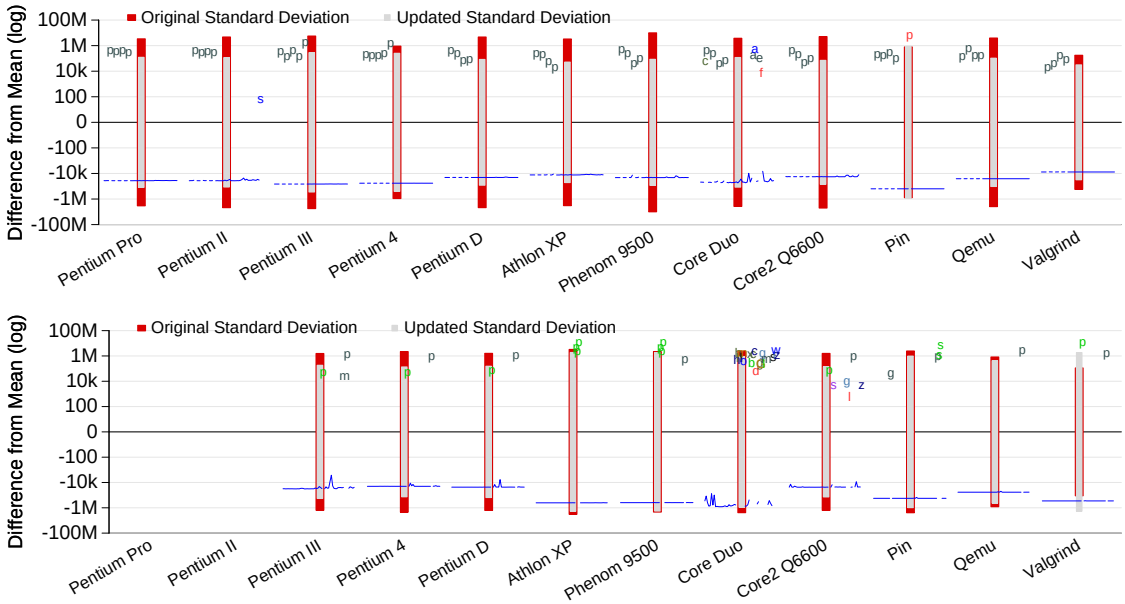

Figure 4.3: Intra-machine results for SPEC CPU2000 (above) and CPU2006 (below). Outliers are indicated by the first letter of the benchmark name and a distinctive color. For CPU2000, the `perlbmk` benchmarks (represented by gray 'p's) are a large source of variation. For CPU2006, the `perlbench` (green 'p') and `povray` (gray 'p') are the common outliers. Order of plotted letters for outliers has no intrinsic meaning, but tries to make the graphs as readable as possible. Horizontal lines summarize results for remaining benchmarks (they're all similar). The message here is that most platforms have few outliers, and there's much consistency with respect to measurements across benchmarks; Core Duo and Core2 Q6600 have many more outliers, especially for CPU2006. Our technical report provides detailed performance information — these plots are merely intended to indicate trends. Standard deviations decrease drastically with our updated methods, but there is still room for improvement.

4.1.5 Inter-machine Results

Figure 4.4 shows results for each SPEC 2000 benchmark (DBI values are shown but not incorporated into standard deviation results). We include detailed plots

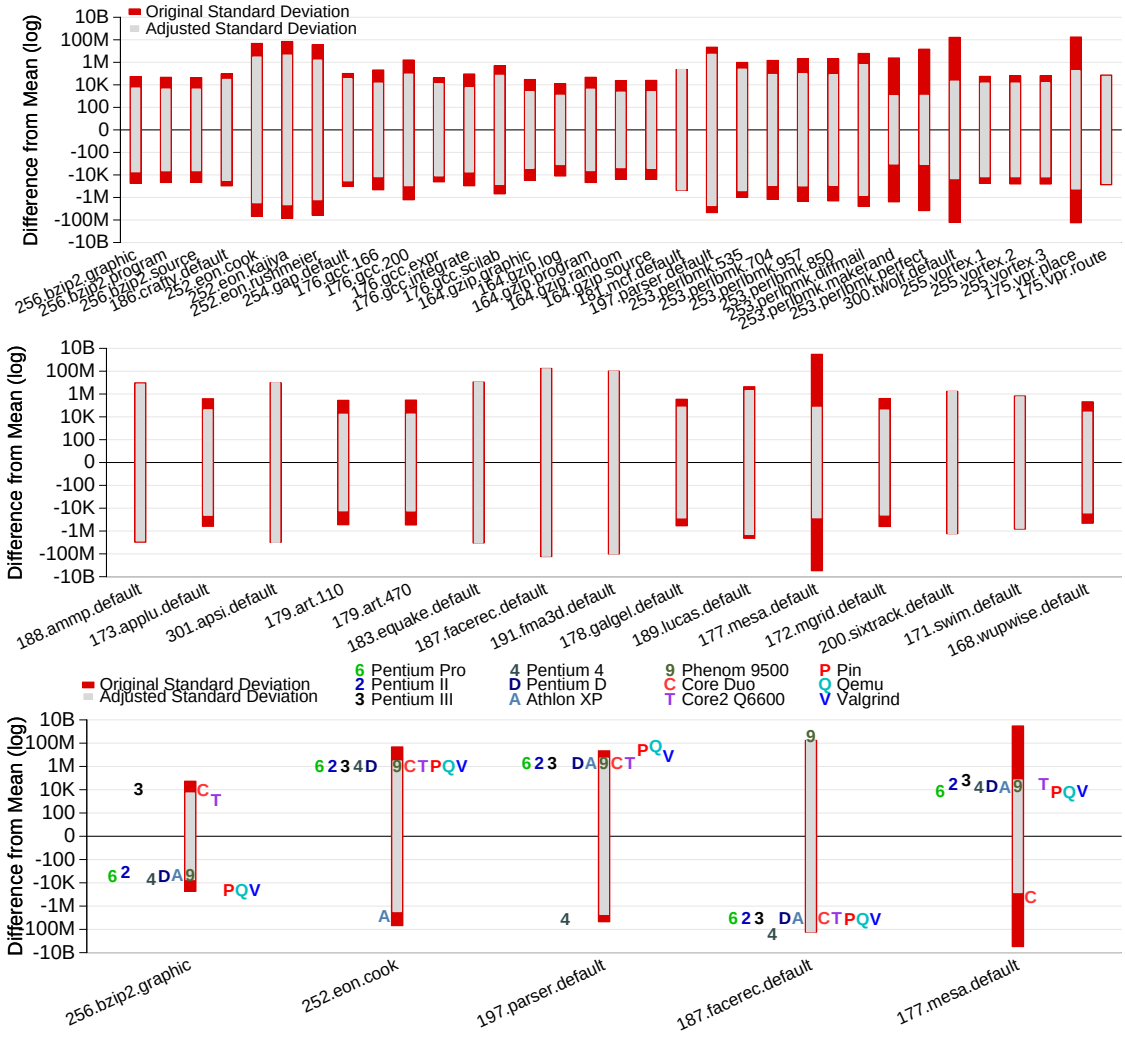

Figure 4.4: Inter-machine results for SPEC CPU2000. We choose five representative benchmarks and show the individual machine differences contributing to the standard deviations. Often there is a single outlier affecting results; the outlying machine is often different. DBI results are shown, but not incorporated into standard deviations.

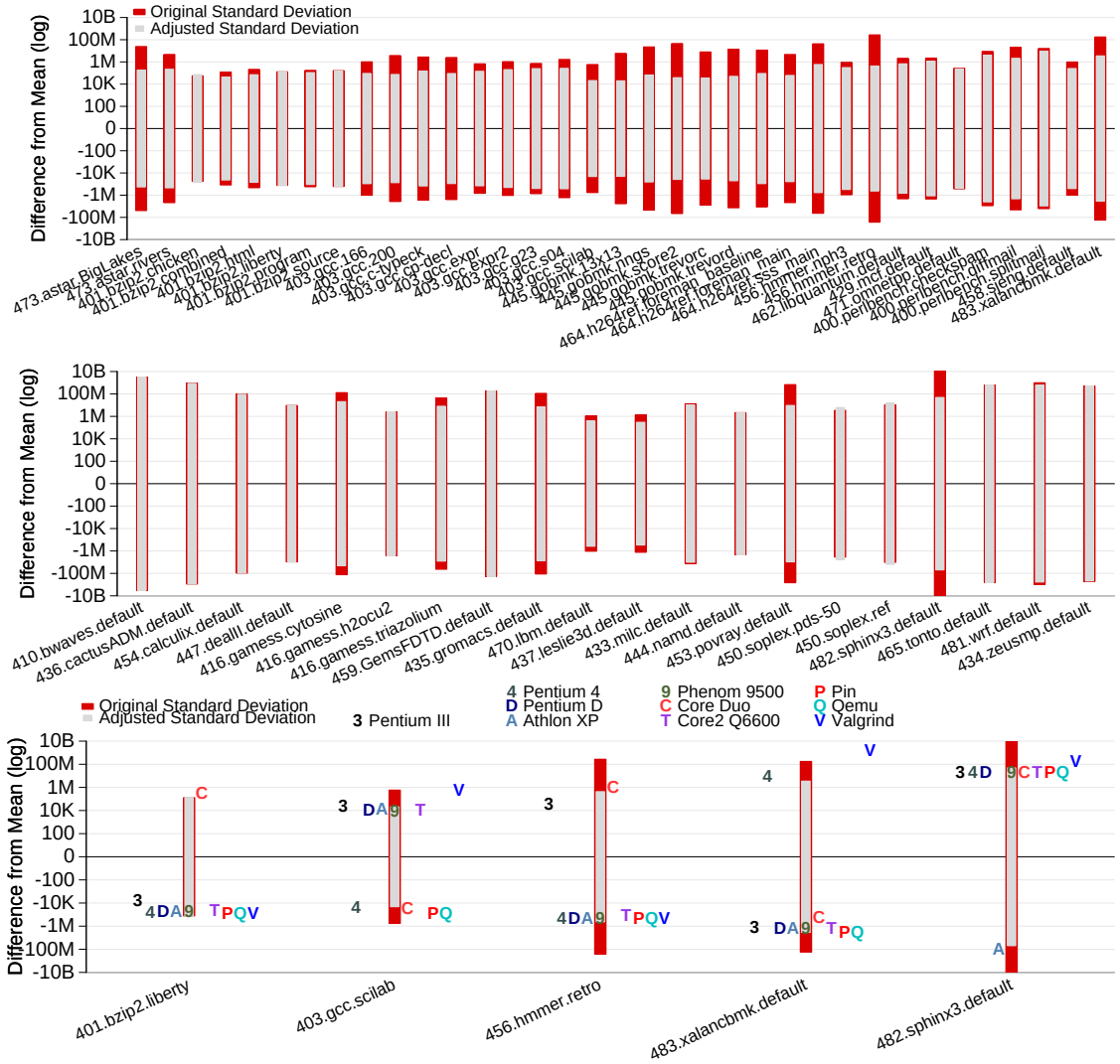

Figure 4.5: Inter-machine results for SPEC CPU2006. We choose five representative benchmarks and show the individual machine differences contributing to the standard deviations. Often there is a single outlier affecting results; the outlying machine is often different. DBI results are shown, but not incorporated into the standard deviations.

for five representative benchmarks to show individual machine contributions to deviations. (Detailed plots for all benchmarks are available in our technical report [154].) Our variation-reduction methods help integer benchmarks more than floating point. The Pentium III, Core Duo and Core 2 machines often over-count instructions. Since they share the same base design, this is probably due to architectural reasons. The Athlon frequently is an outlier, often under-counting. DBI results closely match the Pentium 4's, likely because the Pentium 4 counter apparently ignores many OS effects that other machines cannot.

Figure 4.5 shows inter-machine results for each SPEC 2006 benchmark. These results have much higher variation than the SPEC 2000 results. Machines with the smallest memories (Pentium 3, Athlon, and Core Duo) behave similarly, possibly due to excessive OS paging activity. The Valgrind DBI tool behaves poorly compared to the others, often overcounting by at least a million instructions.

4.2 Deterministic Execution

We found various issues that affect deterministic execution.

4.2.1 Virtual Memory Layout

It may seem counter-intuitive, but some benchmarks behave differently depending on where in memory their data structures reside. This causes much of the intra-machine variation we see across the benchmark suites. In theory, memory layout should not affect instruction count. In practice, both `parser`

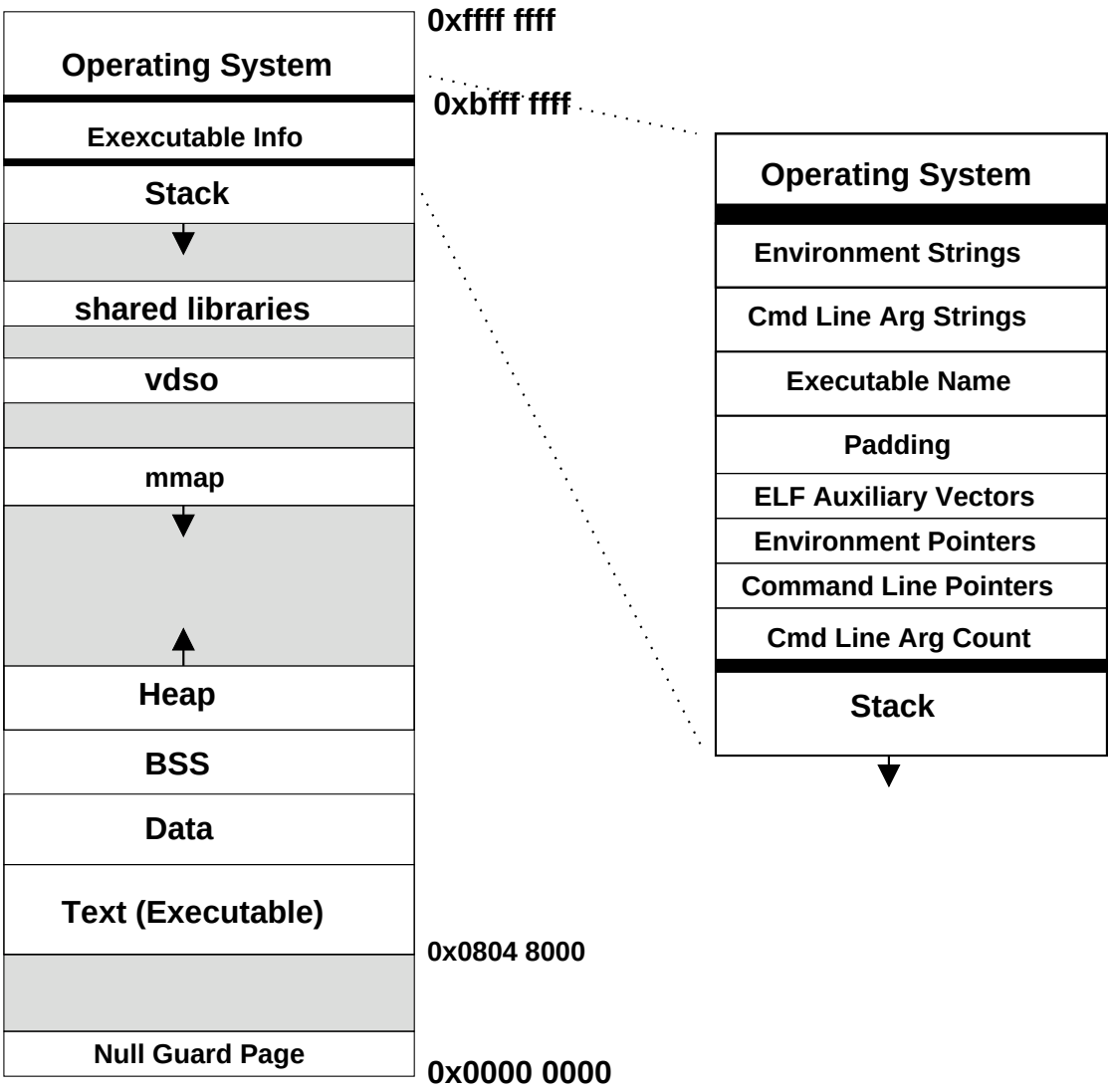

Figure 4.6: The typical layout of virtual memory for a process on 32-bit x86 Linux. If process space randomization is enabled, then the BSS, Heap, mmap and stack can have different offsets.

and `perlbench` exhibit this problem. To understand how this can happen, it is important to understand the layout of virtual memory on x86 Linux (see Figure 4.6). In general, program code resides near the bottom of memory, with initialized and uninitialized data immediately above. Above these is the heap, which grows upward and the mmap region, which on newer kernels grows downward. Near the top of virtual memory is the stack, which grows downward. At the very top of the stack is process information, including command line arguments and environment variables.

Typical programs are insensitive to virtual address assignments for data structures. Languages that allow pointers to data structures make the virtual address space "visible". Different pointer values only affect instruction counts if programs act on those values. Both `parser` and `perlbench` use pointers as hash table keys. Differing table layouts can cause hash lookups to use different numbers of instructions, causing noticeable changes in retired instruction counts.

There are multiple reasons why memory layout can vary from machine to machine. On Linux the environment variables are placed above the stack; a differing number of environment variables can change the addresses of local variables on the stack. The same is true of the executable name (so a program run from a different directory path could change this offset). Also, from kernel to kernel the number of ELF auxiliary vectors changes, and unfortunately these too are above the stack. If the addresses of local variables are used as hash keys then the size and number of any of these executable parameters can affect the total instruction count. This happens with `perlbench`; Mytkowicz et al. [104] document the effect, finding that it causes execution time differences of up to

5%.

A machine's word size can have unexpected effects on virtual memory layout. Systems running in 64-bit mode can run 32-bit executables in a compatibility mode. By default, however, the stack is placed at a higher address to free extra virtual memory space. This can cause inter-machine variations, as local variables have different addresses on a 64-bit machine (even when running a 32-bit binary) than on a true 32-bit machine. Running the Linux command `linux32 -3` before executing a 32-bit program forces the stack to be in the same place it would be on a 32-bit machine.

Another cause of varied layout is due to virtual memory randomization. For security reasons, recent Linux kernels randomize the start of the text, data, bss, stack, heap, and mmap() regions. This feature makes buffer-overrun attacks more difficult, but the result is that programs have different memory address layouts each time they are run. This causes programs (like `parser`) that use heap-allocated addresses as hash keys to have different instruction counts every time. This behavior is disabled system wide by the command:

```
echo 0 >
/proc/sys/kernel/randomize_va_space
```

It is disabled at a per-process level with the `-R` option to the `linux32` command. For our final runs, we use the `linux32 -3 -R` command to ensure consistent virtual memory layout, and we use a shell script to force environment variables to be exactly 422 bytes on all systems.

4.2.2 System Effects

Any Operating System or C library call that returns non-deterministic values can potentially lead to divergences. This includes calls to random number generators; anything involving the time, process ID, or thread synchronizations; and any I/O that might involve errors or partial returns. In general, the SPEC benchmarks carefully avoid most such causes of non-determinism; this would not be the case for many real world applications.

OS activity can further perturb counts. For example, we find that performance counters for all but the Pentium 4 increase once for every page fault caused by a process. This can cause instruction counts to be several thousands higher, depending on the application's memory footprint. Another source of higher instruction counts is related to the number of timer interrupts incurred when a program executes; this is possibly proportional to the number of context switches. The timer based perturbation is most noticeable on slower machines, where longer benchmark run times allow more interrupts to occur. Again, the Pentium 4 counter is not affected by this, but all of the other processors are. In our final results, we account for perturbations due to timer interrupt but not for those related to page faults. There are potentially other OS-related effects which have not yet been discovered.

4.2.3 Sources of DBI Tool Variation

In addition to actual performance counter results, computer architects use various tools to generate retired instruction counts. Dynamic Binary Instrumentation (DBI) is a fast way to analyze benchmarks, and it is important to know how

Table 4.3: Potential overcounted dynamic instructions due to the `rep` prefix (only benchmarks with more than 10 billion are shown).

benchmark	rep counts	% overcount
464.h264ref sss_main	443,109,753,850	15.7%
464.h264ref fore_main	45,947,752,893	14.2%
482.sphinx3	33,734,602,541	1.2%
403.gcc s04	33,691,268,130	18.8%
403.gcc c-typeck	30,532,770,775	21.7%
403.gcc expr2	26,145,709,200	16.3%
403.gcc g23	23,490,076,359	12.1%
403.gcc expr	18,526,142,466	15.7%
483.xalancbmk	15,102,464,207	1.2%
403.gcc cp-decl	14,936,880,311	13.6%
450.soplex pds-50	11,760,258,188	2.5%
453.povray	10,303,766,848	0.9%
403.gcc 200	10,260,100,762	6.1%

closely tool results match actual hardware counts.

The `rep` Prefix

An issue with the Qemu and Valgrind tools involves the x86 `rep` prefix. The `rep` prefix can come before string instructions, causing the the string instruction to repeat while decrementing the `ecx` register until it reaches zero. A naive implementation of this prefix counts each repetition as a committed instruction, and Valgrind and Qemu do this by default. This can cause many excess retired instructions to be counted, as shown in Table 4.3. The count can be up to 443 billion too high for the SPEC benchmarks. We modify the DBI tools to count only the `rep` prefixed instruction as a single instruction, as per the relevant hardware manuals. (Note that older versions of Pin matched real hardware with regards to `rep`, but versions newer than 29972 do not, possibly requiring

extra care when measuring instruction counts).

Floating Point Rounding

Dynamic Binary Instrumentation tools can make floating point problematic, especially for x86 architectures. Default x86 floating point mode is 80-bit FP math, not commonly found in other architectures. When translating x86 instructions, Valgrind uses 64-bit FP instructions for portability. In theory, this should cause no problems with well written programs, but, in practice, it occasionally does. The move to SSE-type FP implementations on newer machines decreases the problem's impact, although new instructions may also be sources of variation.

The `art` benchmark. The `art` benchmark uses many fewer instructions on Valgrind than on real hardware. This is due to the use of the "==" C operator to compare floating point numbers. Rounding errors between 80-bit and 64-bit versions of the code cause the 64-bit versions to finish with significantly different instruction counts (while still generating the proper reference output). This is because a loop waiting for a value being divided to fall below a certain limit can happen faster when the lowest bits are being truncated. The proper fix is to update the DBI tools to handle 80-bit floating point properly. A few temporary workarounds can be used: passing a compiler option to use only 64-bit floating point, having the compiler generate SSE rather than x87 floating point instructions, or adding an instruction to the offending source code to force the FPU into 64-bit mode.

The `dealII` benchmark. The `dealII` SPEC CPU2006 benchmark is problematic for Valgrind, much like `art`. In this case, the issue is more critical: the program enters an infinite loop. It waits for a floating point value to reach an epsilon value smaller than can be represented with 64-bit floating point. The authors of `dealII` are aware of this possibility, since source code already has a `#define` to handle this issue on non-x86 architectures.

Virtual Memory Layout

When instrumenting a binary, DBI tools need room for their own code. The tools try to keep layout as close as possible to what a normal process would see, but this is not always possible, and some data structures are moved to avoid conflicts with memory needed by the tool. This leads to perturbations in the instruction counts similar to those exhibited in Section 4.2.1.

4.3 Summary

Even though originally included in processor architectures for hardware debugging purposes, when used correctly, performance counters can be used productively for many types of research (as well as application performance debugging). We have shown that with some simple methodology changes, the x86 retired instruction performance counters can be made to have a coefficient of variation of less than 0.002%. We have also done some preliminary examinations of retired instruction counts on other architectures, these are available in Appendix C.

CHAPTER 5

32-BIT RISC RESULTS

Cycle-accurate simulators are one of the prevailing modeling tools in computer architecture research. Unfortunately, the results generated by academic "cycle-accurate" simulators can be misleading due to unknown levels of error. More importantly, similar results can often be generated much faster using simulation techniques based on dynamic binary instrumentation (DBI). (Heretofore, we use *cycle-accurate simulations* to refer to tools and results generated in academia. Industry researchers and developers may have much more accurate simulators, but since source code is not generally available to academics, we do not discuss them here.)

In spite of their popularity, cycle-accurate simulators have several drawbacks.

- **Speed:** Simulators are slow, often multiple orders of magnitude slower than native execution. Many researchers commonly use "reduced-execution" methods to compensate, yet these techniques can compound simulation error if not applied carefully. We investigate these methods in detail in Chapter 3.
- **Obscurity:** The simulation tools are rarely used outside the specialized field of computer architecture research. Since the simulators themselves are generally used to run a limited set of benchmark suites, bugs can lurk in the code base.
- **Code Forks:** The code base for an academic simulation tool can quickly become fragmented among the groups using it, or may cease to be maintained entirely. Bugs may be fixed at different times at different institu-

tions. The source codes diverge so much that when a paper claims it uses a particular simulator, that statement may have little meaning, since the code used differs from the mainline (potentially so much so as to be unrecognizable).

- **Generalization:** Simulators are often highly configurable, since the authors usually want a flexible tool that can model a multitude of different hardware configurations. The end result is that a single simulator might model many architectures, but it may not model any particular architecture well. Furthermore, the more flexible a simulator, the easier it is to configure it improperly, often in non-obvious ways.
- **Validation:** Most simulators are not validated against real hardware, and when they are, the results are rarely within 10% error, even after extensive effort to model a known architecture as closely as possible [25, 56, 43]. Exceptions exist, of course, but the most commonly used academic tools have diverged widely from any versions for which validation has been attempted.
- **Documentation:** Simulators are often poorly documented, both at a high level and at the source-code level. This alone probably accounts for more errors in simulation than any overt programming bugs. Researchers simply do not have the information needed to use them correctly.
- **Obsolescence:** Most simulators are already outdated by the time they become mature enough to run useful workloads. It is difficult to gain sufficient documentation on modern processors to accurately implement internals, so well understood but obsolete processors are often modeled, instead.

- **Tools:** Many simulators require a special tool-chain to build suitable executables. The difficulty of using out-of-date toolchains (many need old versions of libraries that are no longer available, for instance) leads to the use of pre-compiled benchmarks that are rarely updated. New advancements in compiler technology are thus lost, since the toolchain is rarely complete enough to compile whole benchmark suites. Some of the more interesting benchmarks may simply be left out due to toolchain difficulties. This is yet another source of error in simulations [35].
- **Operating System:** Many simulators cannot model full operating systems. Cain et al. [31] find that removing the OS from the simulation equation can have a greater impact on results than ignoring effects of speculation.

These problems result in part from the lack of funding for building and maintaining solid academic architectural tools. One or two students cannot create and maintain a tool *and* use it for their doctoral research in a reasonable amount of time, given today's complicated architectures. Many academic researchers end up using an unvalidated or poorly documented simulator modeling a decade-old processor to run only small portions of a decade-old benchmark suite (that was compiled with a decade-old compiler). Needless to say, using such an infrastructure is unlikely to represent "best practices" when performing cutting-edge computer architecture research. Taking that setup and scaling the configuration to match a hypothetical processor only tangentially related to the original design can compound the accuracy problem. Eventually it becomes critical to know how big the potential error is; a small average speedup of 5-10% (which is often sufficient for publication) might, in reality,

be dwarfed by cumulative errors of the infrastructure. [1] To that end, we configure one commonly used cycle-accurate simulator to model a MIPS R12000, and compare simulation versus machine results for five performance metrics. To better understand the tradeoffs between types of simulation tools, we then compare machine results to simulation results generated by a dynamic binary instrumentation tool based on Qemu.

5.1 SESC Cycle-accurate Simulator

SESC [125] is a widely used cycle-accurate simulator. It can simulate CMP systems, but for comparison purposes, we only model a single-core system. The simulator was originally built to model out-of-order MIPS processors, and thus it runs MIPS binaries. It uses an elaborate configuration file that can specify architectures very different from the initially modeled platform. No documentation of peer-reviewed validation is publicly available for SESC. The documentation distributed with the simulator includes a `README.validation` file showing that results for a few microbenchmarks match hardware execution times within about 20% for R10000 and R4400 MIPS-based machines.

We configure SESC to match our reference platform as closely as possible (this required the help of the tool's original author), which turns out to be difficult, despite our machine's being almost exactly the same as the simulator's original design point. Major differences are that the R12000 has a unified 2-page, 64-entry software-controlled TLB (SESC apparently only handles separate data and instruction TLBs), and the R12000's off-chip L2 cache with a way-predictor

[1] We do not discuss issues involved with averages chosen to represent simulation statistics, but see John Mashey [92].

Table 5.1: Configuration of SGI Octane2 machine used for comparison

Processor	300MHz R12000 out-of-order, 4-issue 33 arch registers 64 physical registers
Memory Subsystem	L1i: 32kB, 2-way, 64B L1d: 32kB, 2-way, 32B L2 : 2MB, 2-way, 128B 2GB SDRAM, 1.0GB/s
Branch Predictor	2048 entry 2-bit
TLB	Unified 64-entry

(which can affect L2 cache latencies in a way not easily modeled with SESC). The branch predictor in the R12000 is deceptively non-trivial, and again it is not possible to model exactly. (Many of the arcane architectural details are not sufficiently documented for any simulator author to model exactly without "inside" industrial information.)

We make a best attempt to configure SESC properly. The configuration format is poorly documented, and many necessary options are not described. Sample configurations lack necessary information, and source code is not well commented. In the end, after we spent much time carefully researching and crafting our configuration file, SESC's author found 40 errors. This does not bode well for others attempting to configure the tool without input from SESC authors. The configuration file we used can be found in Appendix L.

We use a default version of SESC, checked out from the CVS server on 7 April 2008 and compiled with gcc version 4.2.4. We use the `-k0x800000 -h0x23400000 -p2` command line options when running benchmarks.

5.2 Reference Hardware

Our reference platform is an SGI Octane2 [156] with an R12000 MIPS processor [167, 111]. A summary of key features is listed in Table 5.1. The machine runs Linux 2.6.22 patched to provide Octane support. The kernel is modified to include the perfmon2 [51] performance counter infrastructure.

The R12000 allows the processor's branch prediction method to be configured at runtime (it is unusual for a processor to be that configurable). We create a custom kernel module (available in Appendix K that sets the proper Branch Diagnostic Register bits (cp0 register 22) to change the branch prediction method on the fly. The processor defaults to a 2048-entry two-bit saturating counter dynamic prediction scheme. This can be changed to various static schemes: always taken, always not-taken, and forward/taken-backward/not-taken. A global pattern history table with a configurable number of bits can be enabled, and the Branch Target Address Cache (BTAC) and Branch Return Cache (BRC) can be individually disabled.

We run microbenchmarks to verify that the performance counters work properly. We use `pfmon` [51] to collect performance statistics. This tool enables performance monitoring by a separate process, so the bookkeeping is handled entirely by the OS kernel, inducing very little user-space overheard. Counts are collected in aggregate for the full program, with no sampling.

There has been concern about the accuracy of MIPS performance counters: Korn et al. [78] find up to 25% error with some counters on the R12000 and R10000 under SGI IRIX. We do not detect similar error; potentially, the differences they see are due to their use of sim-outorder as a reference, which Desikan

et al. [42, 43] found to have similar levels of error.

5.3 DBI-based Simulator

We use Qemu [18] to generate traces consumed by a set of small independent simulators. Qemu uses dynamic retranslation at the basic-block level to convert from one architecture (in this case MIPS) to another (in this case x86). We add code hooks to output needed trace data.

For cache simulation we use the Dinero IV [48] Cache Simulator. Qemu passes trace information in the Dinero file format over a named-pipe to Dinero (which runs in a separate process). To determine branch prediction information we write a custom branch predictor (source available on our website). This predictor runs in a separate process and obtains the full instruction stream (both address and instruction value) from Qemu over a named-pipe. The predictor decodes MIPS instructions and determines which are branches (taking special care to handle the "predict taken" `beql` instructions properly). A branch is determined to be taken or not by buffering an additional two instructions to see if the address after the delay slot is PC+8.

Because each of our tools runs in a separate process, we can take advantage of CMP and SMP systems better than most cycle-accurate simulators. Each process can live on its own core, and running the branch predictor thread at the same time as the cache thread adds negligible overhead on a four-processor machine. The limiting factor here is the cache simulator, not dynamic translation and execution of the binary.

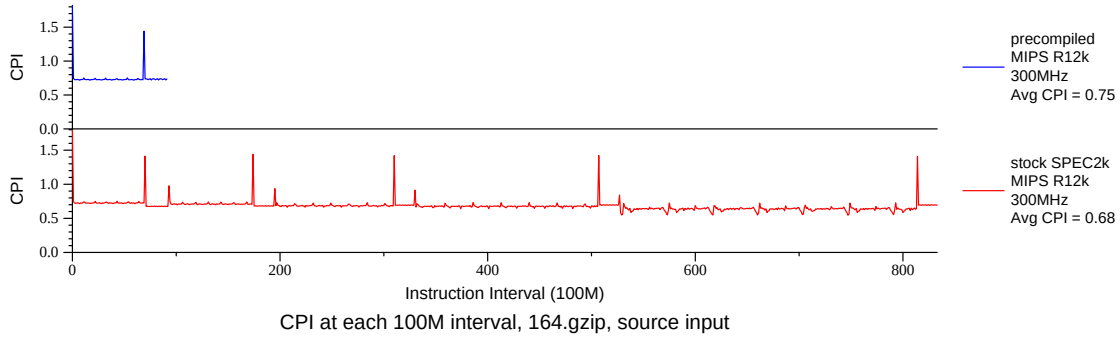

Figure 5.1: The precompiled SPEC 2000 benchmarks available from the SESC website have potentially been modified to reduce run-time. A phase chart gathered with hardware performance counters shows behavior of the provided precompiled binary on top and that of a binary we compiled from original SPEC sources (with gcc) on bottom.

5.4 Benchmarks

To evaluate the various simulation methods, we use SPEC CPU2000 [136] benchmarks. To enable comparison with past uses of the SESC simulator, we use the pre-compiled versions of the benchmarks provided on the SESC website. All three of our test platforms can run these benchmarks unmodified.

Unfortunately the pre-compiled benchmarks have some limitations. Although not documented as such, they are not plain CPU2000 binaries. Extra `printf()` commands have been scattered throughout the code (presumably for debugging purposes or for controlling partial simulation experiments), and some benchmarks have been modified for faster run times. As an example, see Figure 5.1, which shows that `gzip` — as provided — only executes a small fraction of the full benchmark. In addition, not all of the CPU2000 benchmarks are included with the precompiled binaries. We run full reference input sets for all

Table 5.2: Comparison of simulation times

Method	Fastest		Slowest		Mean Slowdown
R12000	15s	(gzip.log)	57m23s	(swim)	–
QEMU	13m52s	(gzip.log)	1d20h20m47s	(sixtrack)	38x
SESC	2h17m38s	(gzip.log)	16d02h53m15s	(mgrid)	393x

experiments.

5.5 Results

We run as many SPEC 2000 benchmarks as possible on the various platforms. Relative run times are shown in Table 5.2. For the simulated results, we run on a large cluster of 4-processor 3.46GHz Pentium D nodes, each with 4GB of RAM.

5.5.1 Absolute Results

Figure 5.2 shows actual and predicted L1 instruction cache miss rates. Our three methods calculate instruction cache misses in different ways. For the performance counter results, these graphs show decoded instructions versus instruction cache misses; for SESC and Qemu the graphs show graduated instructions versus instruction cache misses. The number of instruction cache misses in the floating point case is so small that a small absolute error can cause a large percentage error. Qemu has problems with the `art` benchmarks, which we are investigating.

The reference system has write-back caches, which can introduce accuracy

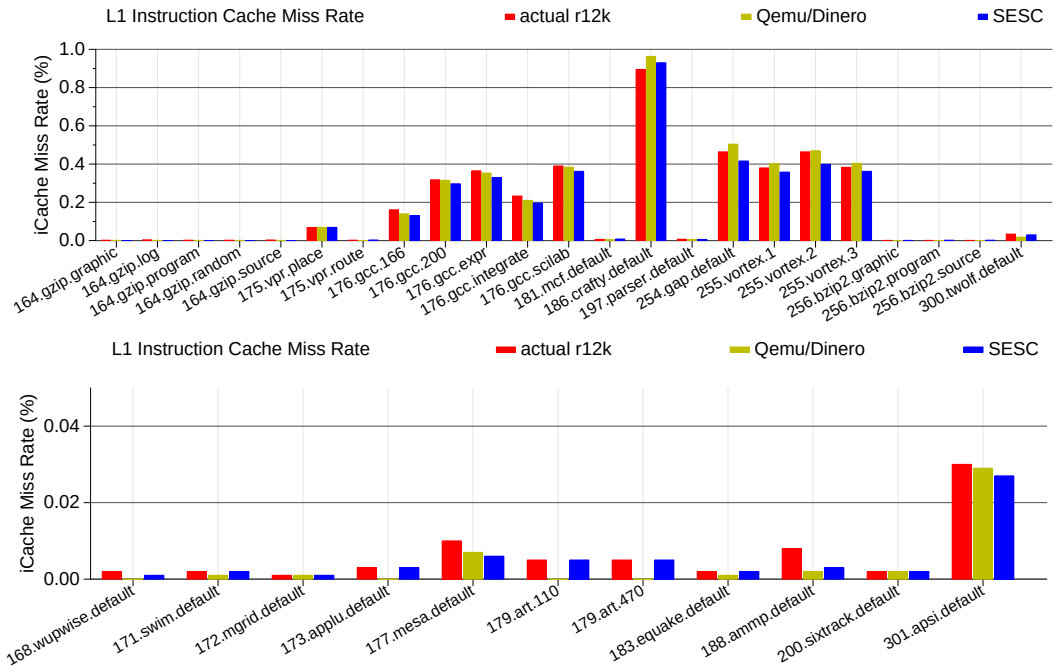

Figure 5.2: Instruction cache miss rate with integer benchmarks above and floating point below.

issues with the performance counters. Memory accesses that occur while the benchmark process is not running can change values in the cache. While we attempt to run the benchmarks on an otherwise quiet system, other processes and even the operating system can evict cache lines on the real system in ways that cannot be modeled in the simulator. Similarly, values stored into cache may not be accounted for by the performance counters if the actual write-back to memory happens when in a different processor context.

Qemu does not follow wrong-path execution [2] , which can account for some of the differences from actual hardware. Likewise, SESC does not follow wrong-path execution; the code path that models speculation is out of date, and is thus disabled in the default configuration. Despite not executing wrong-path

[2]There has been work done to enable wrong-path execution support on Qemu [33, 32] but the code involved has not been released.

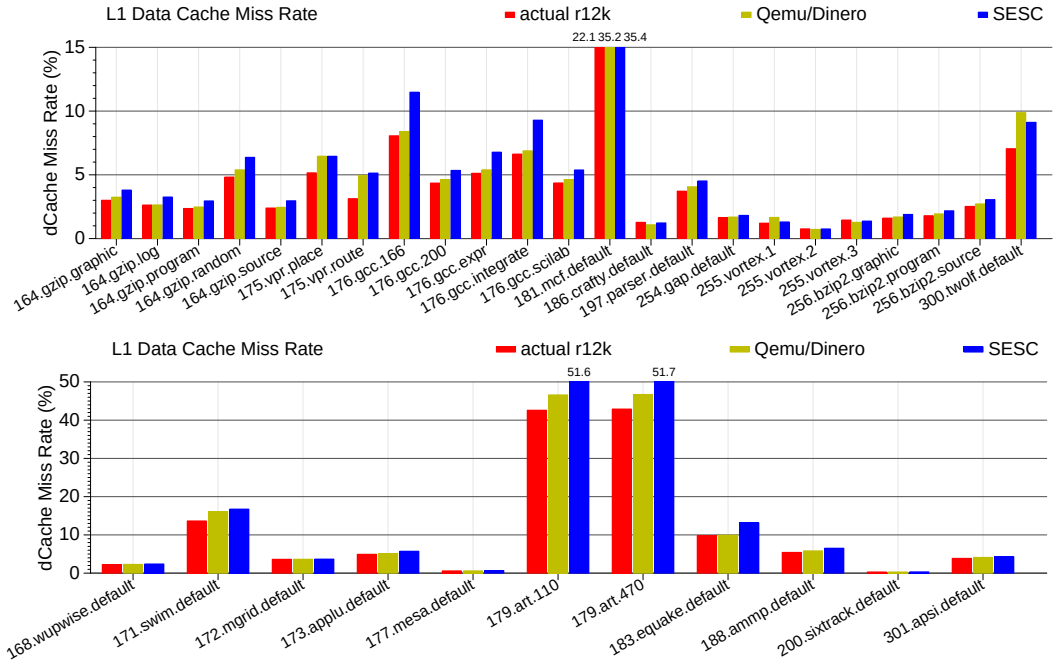

Figure 5.3: L1 data cache miss rate with integer benchmarks above and floating point below.

instructions, results are quite accurate; this shows that full cycle-accuracy is not always needed to generate good cache simulation results (and further supports the conclusions of Cain et al. [31] regarding OS impact versus speculation, at least in the case of Qemu).

Figure 5.3 shows L1 data cache miss rates, and Figure 5.4 shows L2 miss rates. The latter is important, since L2 cache misses must traverse the processor bus of a multiprocessor system. If the tool used records vastly incorrect numbers of misses, multiprocessor simulations will generate erroneous data that could influence a final design. SESC generally does poorly predicting L2 miss rates for floating point benchmarks. This could indicate that the floating point pipeline sections of the configuration file need further adjustment.

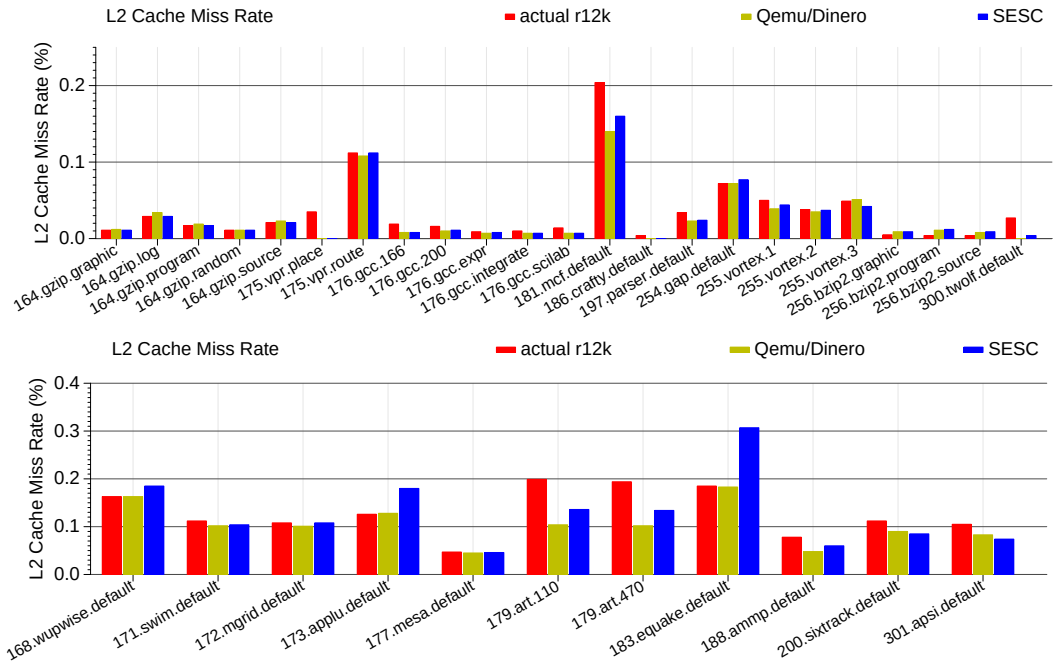

Figure 5.4: L2 cache miss rate with integer above and floating point below. None of the simulations captures `mcf`'s behavior well. None of the simulation methods predicts the `art` benchmarks well.

The R12000 has a complicated off-chip cache. In order to save pins, the machine incurs significant overhead in changing cache ways. To mitigate this, it uses a cache way-predictor, with a penalty on a miss. None of the simulators model this aspect of the system, which can potentially become another source of modeling error.

Figure 5.5 shows branch predictor results. The R12000 can predict and fetch past up to four branches, so many speculative instructions can be in flight. Qemu and SESC cannot model this. In fact, the R12000 branch predictor has many hardware subtleties that neither Qemu nor SESC can model.

Figure 5.6 shows CPI results. Qemu does not model time, so we approximate cycles with the formula:

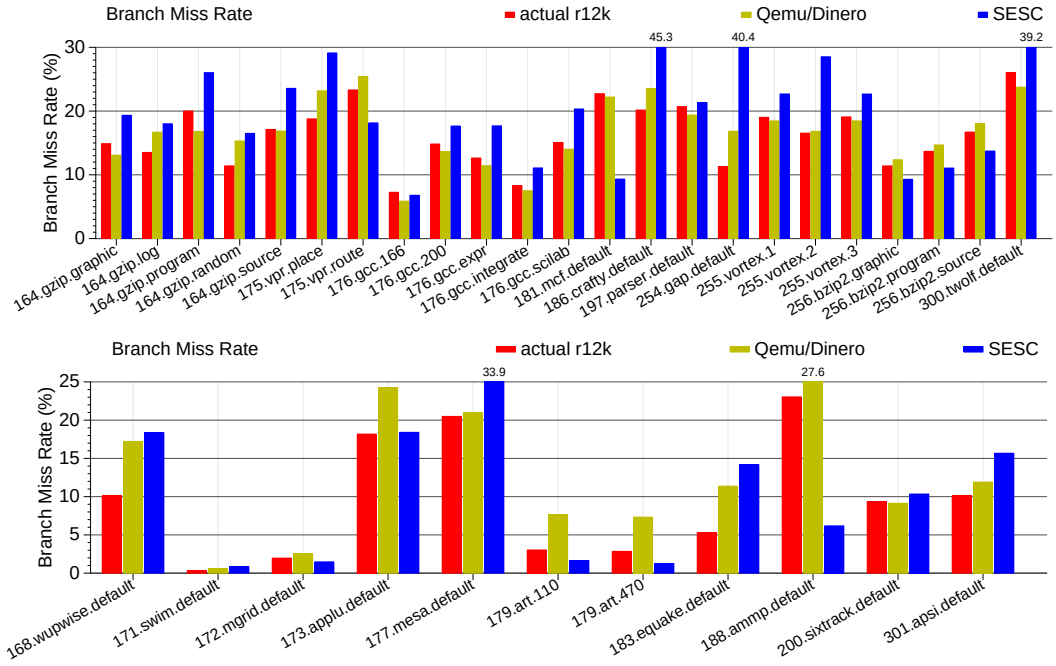

Figure 5.5: Branch miss rate with integer above and floating point below. The hardware can have up to four outstanding branches; Qemu and SESC do not model wrong-path execution.

$$cycles = \frac{I_g * L1_{ht}}{ifs} + DL1_a L1_{ht} + L1_m L1_{mt} + L2_m L2_{mt} + Br_m Br_{mt}$$

where I_g is graduated instructions, $L1_{ht}$ is L1 hit time (2 cycles), ifs is the instruction fetch size (4 words), $DL1_a$ is L1 data accesses, $L1_m$ is L1 misses, $L1_{mt}$ is L1 miss time (14 cycles), $L2_m$ is L2 misses, $L2_{mt}$ is L2 miss time (120 cycles), Br_m is number of branch misses, and Br_{mt} is branch miss delay (2 cycles)

This is an empirical model that was arbitrarily chosen because it seems to match well against the parameters we have. It is similar in idea to CPI generation functions for the R10000 presented by Luo et al. [88]. The L1 icache parameter might be spurious; its primary effect is to limit the minimum IPC to two, which is what is found on the SPEC benchmarks. In theory the R12000 can have an IPC of up to 5; more investigation is needed to explain this discrep-

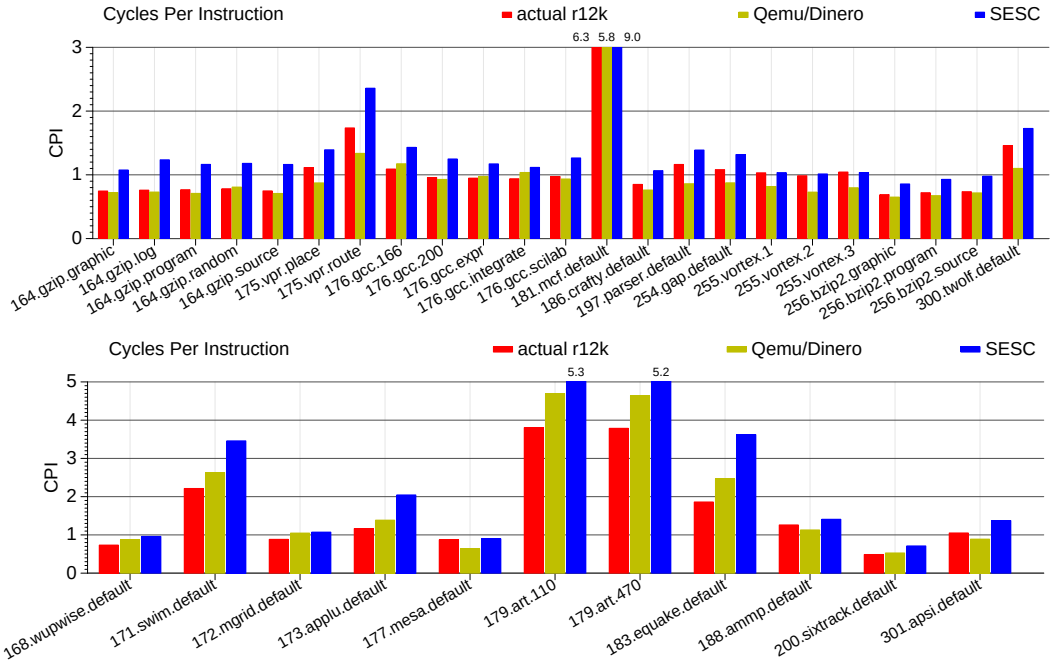

Figure 5.6: CPI results with integer above and floating point below.

ancy. The data cache misses should be hidden by out-of-order execution too, although depending on the memory subsystem design this might not happen. Luo et al. [88] found up to an 80% stall rate for one configuration of an R10000 processor.

CPI is the metric most often used in validation, so it is important to have these values match hardware as closely as possible. There are many architectural and software causes of cycle variation not modeled by either simulator. Most notably, Operating System effects are not modeled.

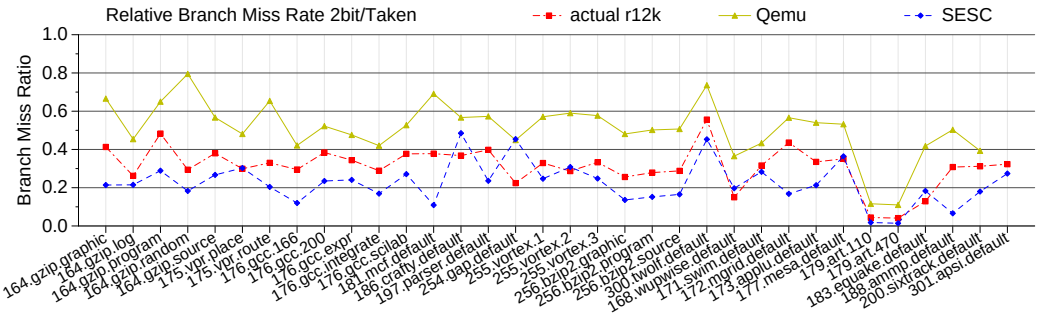

Figure 5.7: Always taken branch predictor miss rate, normalized against dynamic two-bit results.

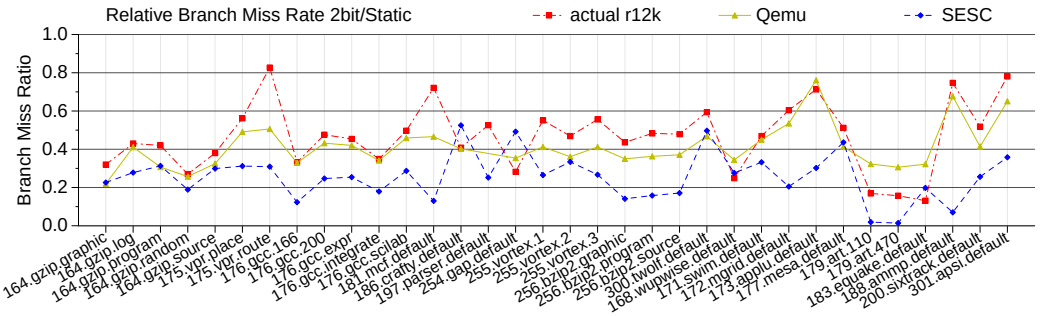

Figure 5.8: Static branch predictor miss rate, normalized against dynamic two-bit results.

5.5.2 Relative Results

Many researchers hold that absolute results are not as important with cycle-accurate simulation, but that relative results are what matter most. As long as the trends are consistent, then a simulator is still useful, even if the simulator is unvalidated and the error is large. To investigate this, we configure our R12000 to use different branch predictors. We plot relative differences in the metrics to see if consistent trends are visible.

Figure 5.7 shows the relative reduction in branch predictor miss rate when going from a dynamic two-bit predictor to an always-taken predictor. The figure

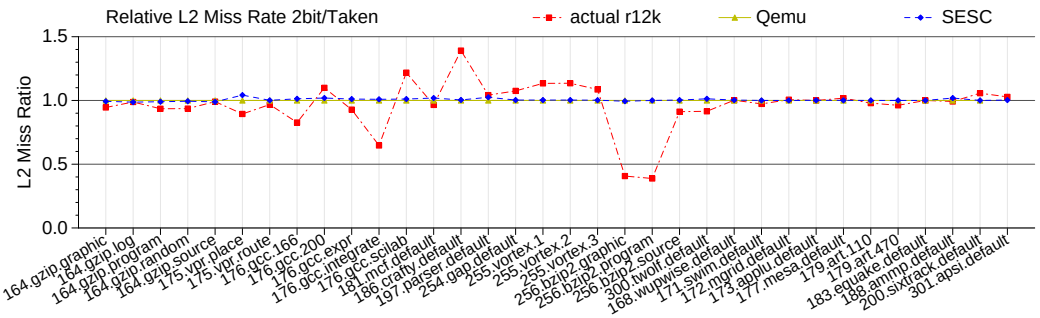

Figure 5.9: L2 cache miss rates with the always-taken predictor, normalized against two-bit results.

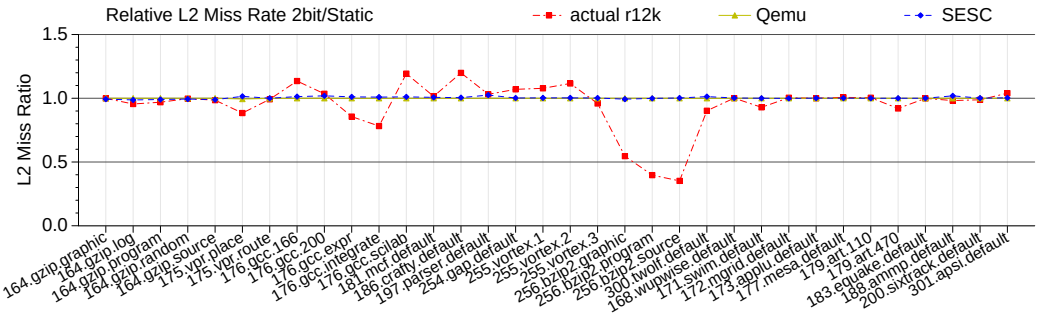

Figure 5.10: L2 cache miss rates with the static predictor, normalized against two-bit results.

shows that trends are similar across all benchmarks, although Qemu results are optimistic and SESC results are pessimistic. Figure 5.8 compares a static backward/taken forward/not-taken predictor to the dynamic two-bit predictor.

Figure 5.9 shows how the always-taken predictor affects the L2 cache miss rate compared to the two-bit predictor. Neither Qemu nor SESC models wrong-path execution, so they exhibit identical memory access behavior even with different branch predictors. Neither simulation method can predict the significant predictor-based changes in L2 behavior observed on actual hardware. Results for the forward/backward static predictor, shown in Figure 5.10, are similar.

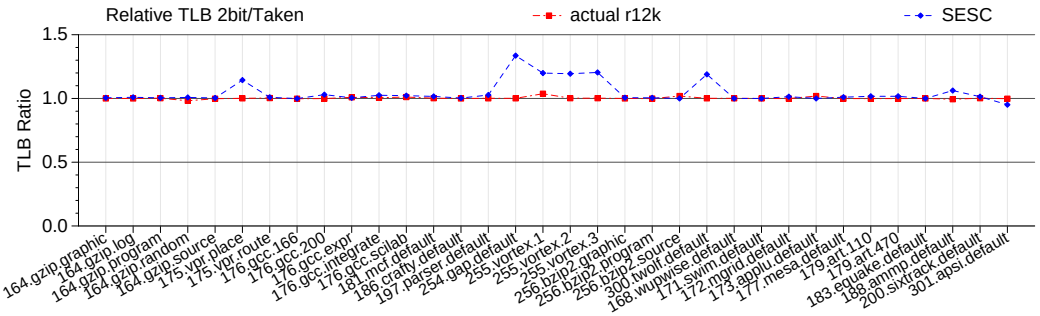

Figure 5.11: TLB misses with always taken, normalized against two-bit.

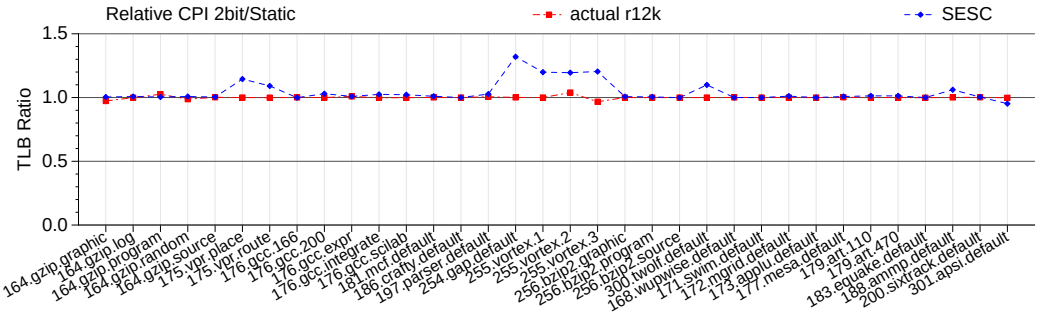

Figure 5.12: TLB misses with static predictor, normalized against two-bit.

Figures 5.11 and 5.12 show TLB behavior. Results are not shown for Qemu because a trace-based TLB simulator was not available. On actual hardware, the branch predictor seems to have minimal impact on TLB behavior. The MIPS TLB is managed in software, usually with random replacement. This means that it is easy for results to diverge. Also, MIPS has a unified instruction/data TLB, which SESC cannot model.

Figure 5.13 and Figure 5.14 show the relative results for CPI. Qemu results are close to those for the R12000, despite the cycle counts being based solely on cache and branch predictor miss rates.

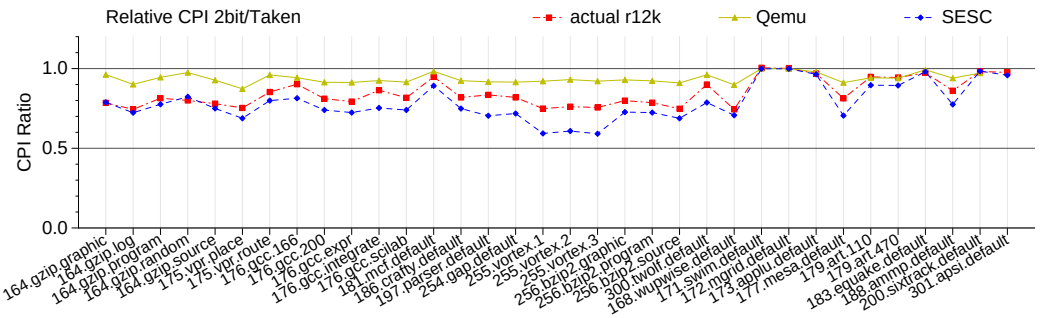

Figure 5.13: CPI with always taken normalized against two-bit results.

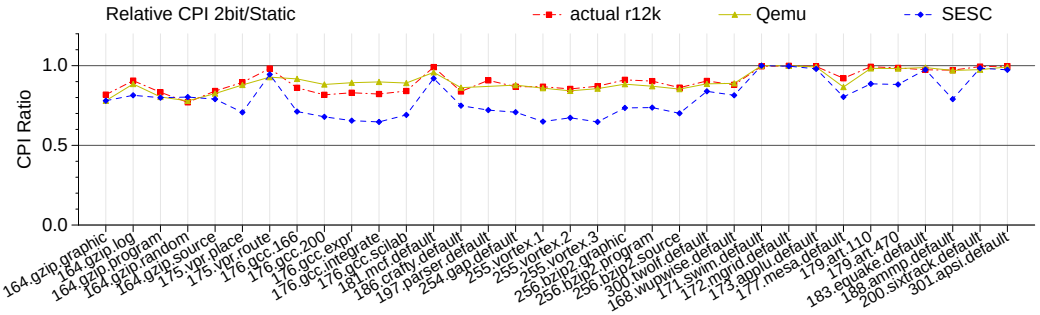

Figure 5.14: CPI with static predictor normalized against two-bit results.

5.5.3 Summary

A summary of the absolute results is shown in Table 5.3. The weighted average of the various metrics is taken across all benchmarks that run to completion on all three platforms. This is a total of 22 benchmarks (19 integer, 3 floating point) which, unfortunately, only represents a portion of the 48 SPEC CPU2000 benchmark/input pairs. SESC does not perform noticeably better than Qemu, despite taking an order of magnitude longer to run.

Table 5.4 shows the percent error of the average relative performance differences. The CPI results show that these methods can be used to predict performance with an average error of 15%. The L2 Cache results show that sometimes

Table 5.3: Summary of results. The weighted average is across all of the SPEC 2000 benchmarks which ran to completion on all three platforms: 23 integer and 11 floating point (this is unfortunately only a portion of the 48 available benchmark/input combinations).

Metric	Bench Type	R12000 Weighted Average	Qemu Weighted Average	Qemu % Error	SESC Weighted Average	SESC % Error
L1I$ Miss Rate	Int	0.233%	0.334%	43.5%	0.248%	6.4%
	FP	0.008%	0.001%	-83.9%	0.006%	-23.9%
L1D$ Miss Rate	Int	3.928%	4.260%	8.5%	4.726%	20.3%
	FP	5.230%	6.406%	22.5%	6.485%	24.0%
L2$ Miss Rate	Int	0.058%	0.051%	-11.9%	0.042%	-27.6%
	FP	0.127%	0.107%	-16.2%	0.128%	0.4%
BrPred Miss Rate	Int	18.9%	18.4%	-2.7%	27.0%	42.9%
	FP	12.7%	18.2%	43.2%	15.0%	18.4%
CPI	Int	1.20	1.03	-14.6%	1.47	22.6%
	FP	1.09	1.41	29.3%	1.60	46.4%

Table 5.4: Summary of relative results. The relative results compare the relative results when moving from 2-bit branch predictor to either taken or static. The error shown is the relative error between the relative average means of all benchmarks on actual hardware versus the predicted relative average means of the simulated results. The results represent the 33 of the SPEC CPU 2000 benchmarks which ran to completion on all three platforms.

Metric	Brpred Type	Qemu % Error	SESC % Error
BrPred Miss Rate	Taken	64.1%	-28.0%
	Static	-11.0%	-44.9%
L2$ Miss Rate	Taken	5.6%	6.1%
	Static	7.1%	7.4%
CPI	Taken	11.5%	-7.1%
	Static	0.1%	-10.9%

results can be deceptive; even though neither QEMU nor SESC models wrong-path execution, results still fall within 10% error for relative L2 cache miss rate.

CHAPTER 6

64-BIT CISC RESULTS

Our work in Chapter 5 finds acceptable results when using DBI methods to simulate an obsolete RISC processor; we extend this work to a more modern 64-bit x86 platform. Memory access patterns on modern CISC (Complex Instruction Set Computer) systems differ from older RISC systems, with variable-sized instructions, aggressive prefetching, and SSE vector-like memory accesses. Unfortunately CISC simulations run slower than RISC. The exact slowdown depends on the simulator, but on the m5 simulator moving from Alpha to x86 has a slowdown of at least a factor of two.

6.1 RISC/CISC differences

RISC chips, even sophisticated ones such as the MIPS R10000 or Alpha 21264 (as simulated by common simulators), are missing many features found in entry-level x86 processors.

Here are some CISC "features" that most RISC implementations do not have to worry about:

- Unaligned instructions
- Variable length instructions
- Instructions that cross cache lines
- Complicated lock instructions
- Complicated string instructions
- Hardware square-root and transcendental functions

- μop Decoder Cache
- Complex μop issue logic, "fusing"
- Self-modifying code
- Micro-code assist on complex instructions (NaN, Denormals, Div/0, Underflows)

6.2 Modern CPU Features

Cycle-accurate simulators tend to model older implementations of architectures. Modern architectural features are often left out of a simulator as they do not affect correctness, but can affect behavior. Modern implementations of RISC chips (such as ARM, MIPS, Power and SPARC) might have these features, but many simulators do not support them. Recent x86 binaries make use of these features, and since comprehensive x86 simulators are a recent development, the simulators have to handle these newer features to run the binaries properly.

There are many features that can affect architectural simulation but are not commonly found in simulators:

- Vector instructions (most modern RISC architectures have support, but are not commonly used).
- Hardware prefetch
- Various software prefetch types (including non-temporal)
- Large pages (2MB, 1GB)
- Memory disambiguation predictor

- Execute small loops out of instruction fetch unit (without accessing cache): LSD Loop Stream Detector
- Trace caches
- Thermal trip support
- CPU frequency scaling
- MTRR/PAT Page attributes (set cache behavior at page level)
- ECC memory
- Return address prediction
- Stack pointer prediction
- Sophisticated branch prediction schemes
- Complicated memory hierarchies
- On-chip memory controllers

6.3 μop Concerns

The x86 architecture does not directly execute complex CISC instructions. During fetch and decode these complex instructions are broken down into RISC-like instructions known as μops.

Since μops are "RISC-like", RISC simulators can be repurposed to act as backends for CISC simulators. This is a common simulation methodology [53, 23, 129, 141, 134, 26], that as far as we know has not been validated.

Figure 6.1 shows L1 data cache accesses per μop on MIPS and three x86_64 architectures for the `gzip.program` benchmark (complete μop phase diagrams can be found in Appendix H). We measure μop counts using the counters listed in Table 6.1.

Table 6.1: Hardware performance counters used for μop experiments

machine	Retired Instructions	Retired μops
Phenom	retired_instructions	retired_uops
Core2	instructions_retired	uops_retired:any
Pentium D	instr_completed:nbogus	uops_retired:nbogus
Pentium Pro	inst_retired	uops_retired
Atom	instructions_retired	uops_retired:any

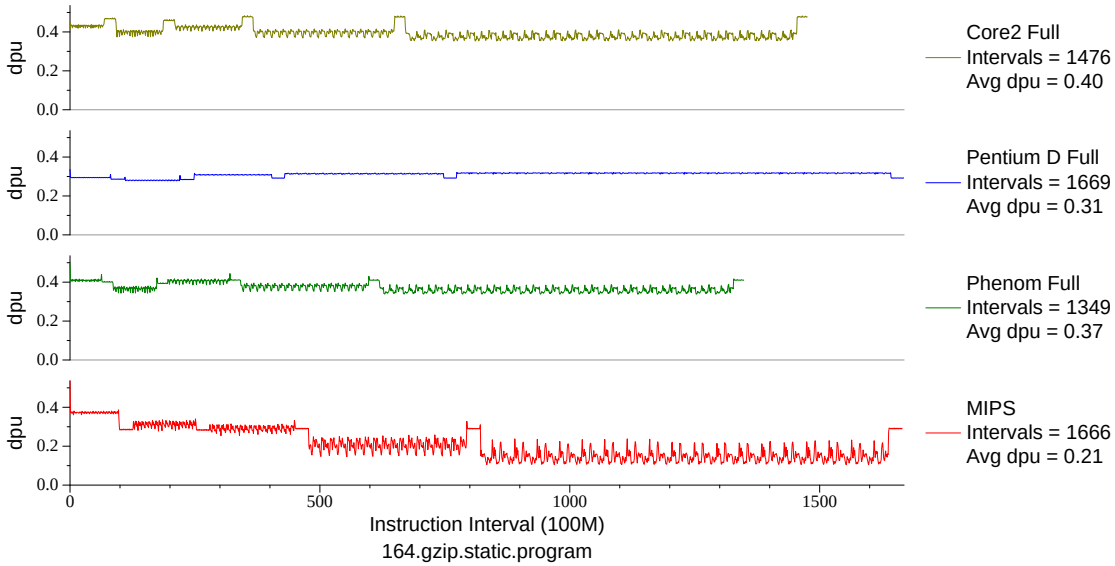

Figure 6.1: Data cache accesses per μop for `gzip.program`

An unexpected result is that the μop behavior varies between implementations of the same architecture. The set of μops is not fixed and architects are free to change it at any time. Figure 6.1 shows that the MIPS instruction trace would make a believable x86_64 μop stream, however it does not closely match any of the existing machines. Care should be taken when using RISC results as a μop substitute.

Figure 6.2 is an overall summary of μop versus instructions differences for

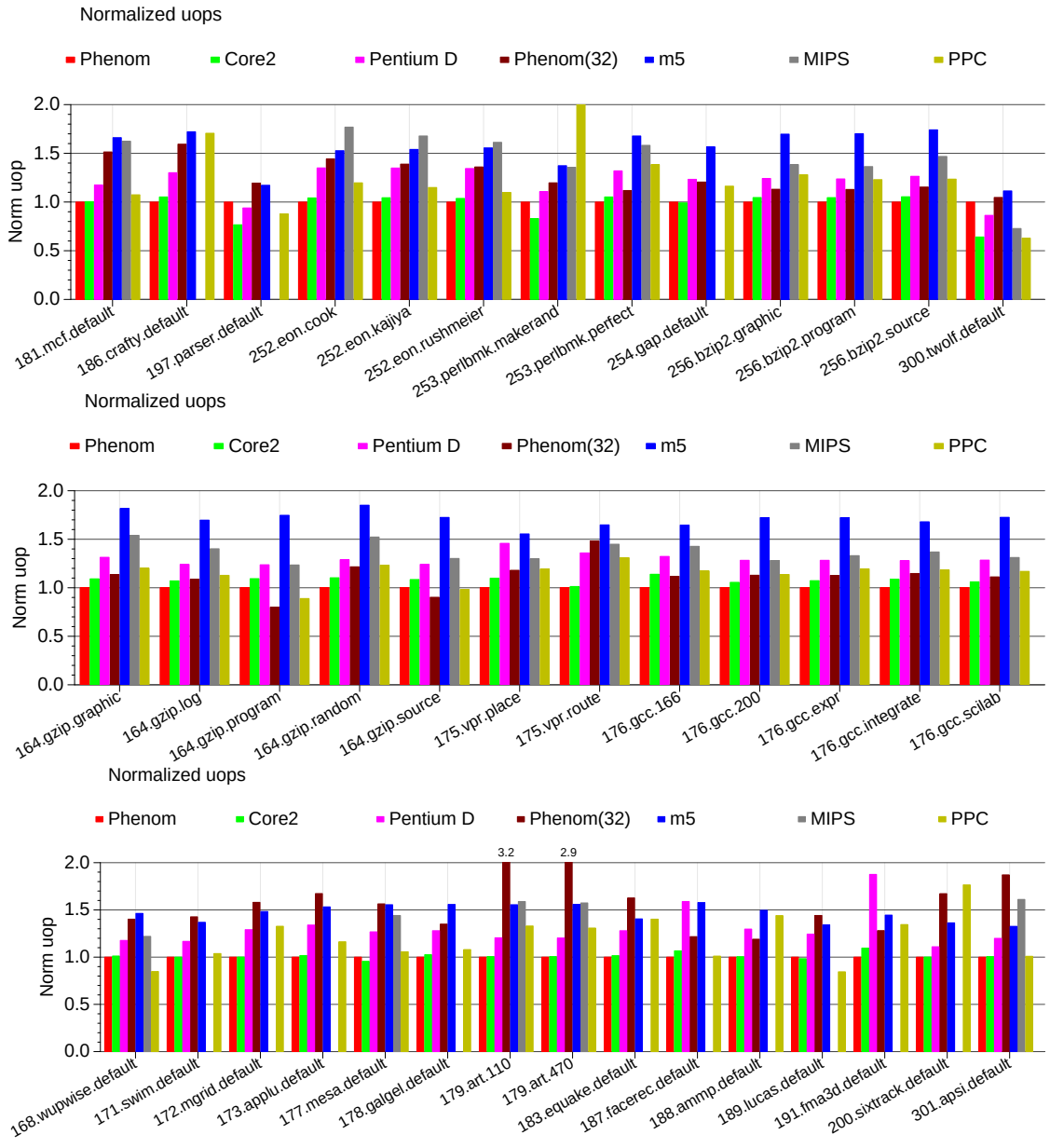

Figure 6.2: Normalized μops per benchmark for three x86_64 implementations, a 32-bit x86, the m5 simulator, and two representative RISC architectures.

Table 6.2: Number of uops required for an assortment of x86 instructions

instruction	Phenom	Core2	Pentium D	Pentium Pro	Atom
add %eax,%edx 32-bit int add	1	1	1	1	1
add mem,%eax 32-bit add from mem	1	1	2	2	1
imul %eax,%edx 32-bit int multiply	2	3	2	3	3
rep stosb repeated string store	0.3	0.43	0.55	0.6	3
fadd 1.0,pi floating point add	23	1	1	4	1
fsincos floating point sincos	60	101	150	107	118
haddps 128-bit horizontal add	1	6	3	N/A	5
pslldq 128-bit shift	1	2	1	N/A	1

all of the SPEC CPU2000 benchmarks. The relative number of μops varies by benchmark, even on the same architecture. The 32-bit machine has many more μops, especially on floating point benchmarks; this is because the 32-bit program is using x87 floating point, which produces many more μops than the SSE-based floating point used on the x86_64 machines. The two comparison RISC machines are roughly the same as the x86 machines. The m5 counts are in general much too high; this is because the simulator's μop generation has not yet been matched to that of an actual machine.

Table 6.2 breaks out μop counts for a few selected instructions, to show why it is difficult to make generic statements about μop behavior. An additional challenge is that μop counts may vary from run to run, because unlike the retired instruction counters, the μop counts include microcode, exception, interrupt,

and various other effects [10, 72]. There is not always a static mapping between μops and instructions; operations like floating point transcendental functions can take varying numbers of instructions depending on the operands involved.

Another issue with μops is that hardware performance counters do not always measure the same results across architectures. Kenneth Hoste [68] found that some architectures "fuse" the μops, making it difficult to compare results, specifically between Nehalem and Core2 implementations.

Due to all of the issues found with μops, retired instructions may be the best base metric to use when comparing x86 implementations. This might seem counter-intuitive, because it sacrifices some of the fine detail provided by the knowledge of μop behavior.

6.4 Evaluation Methodology

We evaluate x86 simulation using three different methods: the Valgrind DBI tool, the m5 cycle-accurate simulator, and hardware performance counters.

6.4.1 Valgrind DBI-based Simulator

To test DBI-based simulation we use the Cachegrind [112] tool that comes with the Valgrind [113] DBI infrastructure. This tool simulates a configurable single-core cache and also can simulate a simple branch predictor.

We configure the cache simulator to have the same basic cache configuration as the Phenom hardware described in Table 6.3, which means the command line

Table 6.3: Configuration of AMD Phenom machine used for comparison

Processor	2.2GHz Phenom out-of-order, 3-issue 16 arch registers
L1 Instruction Cache	64kB, 2-way, 64B prefetch 2 lines on miss
L1 Data Cache	64kB, 2-way, 64B write-allocate, write-back LRU, ECC, MOESI, 3-cycles
L2 Cache	512kB, 16-way, 64B non-inclusive victim 9-cycles, per-core
L3 Cache	2MB,, 32-way, 64B non-inclusive victim shared by all cores
Main Memory	2GB DDR2 integrated memory controller built-in prefetcher

options

`--tool=cachegrind --cache-sim=yes --branch-sim=yes`

`--I1=65536,2,64 --D1=65536,2,64 --L2=524288,16,64`.

The average slowdown while running Cachegrind is 29x over baseline.

6.4.2 m5 Cycle-accurate Simulator

We use the m5 [22] simulator as a reference cycle-accurate simulator for our study. It is currently one of only two readily available academic simulators capable of running x86 binaries, the other being PTLsim [172].

m5 can simulate multiple architectures, but we are primarily interested in

x86 emulation. m5 can run both standalone statically linked binaries in syscall emulation mode, as well as full operating systems in full system mode. Unfortunately full system mode has not been tested for x86, so we are limited to using syscall emulation mode.

m5's x86 support is new; so new that it was not working when we started this work. We contribute a large number of patches that allowed the SPEC CPU2000 benchmarks to run correctly to completion on the simulator, and most of these patches have been merged into the project. There are still some limitations to x86 support, most notably that x87 floating point is not implemented; only binaries compiled to use SSE instructions will work.

Another issue with m5 is that x86 support is so new that only the simple atomic model of execution is supported. This treats each instruction as a single atomic entity. The detailed (in-order) and out-of-order models are not supported, which limits the experiments that can be run. This is unfortunate, but the only real alternative (PTLsim) has show-stopping issues as well, leaving us with no clear best choice for our experiments.

We configure m5 to match our Phenom machine described in Table 6.3 as closely as possible without requiring code changes to the simulator. This limits our changes primarily to cache parameter settings. We cannot model a branch predictor, as that requires the non-working detailed execution model; the same is true for speculative execution.

We use a development version of m5 checked out of the code repository on 16 November 2009, with patches added that enable full x86 support (mainly some missing syscalls and instruction corner cases). We also add code which

adds extra statistics dumping (to print instruction count as well as μop count, and to dump stats at regular intervals).

The average slowdown of m5 running in simple atomic mode with caches enabled 2882 times slower.

6.4.3 Reference Hardware

Table 3.2 lists the machines used in our experiments. The performance counters we use are listed in Table 6.4.

We primarily use the Phenom (summarized in Table 6.3) for gathering results, as the cache simulator we intend to use supports the MOESI protocol for AMD-style machines. The Phenom has a complicated memory hierarchy. It has a 64KB, 2-way, 64 byte linesize, L1 instruction cache; on a miss it assumes temporal locality and fetches two lines, the missing line and the one following. It has a 64KB, 2-way, 64 byte linesize, L1 data cache which is write-allocate, write-back, ECC and an LRU replacement policy. Cache coherence is maintained with a MOESI-like protocol, and there is a latency of 3-cycles. The L2 Cache is per core, 512KB, 16-way, 64 byte linesize, non-inclusive victim, with a latency of 9-cycles. The L3 Cache is system wide, 2MB, 32-way, 64 byte linesize, which behaves as a non-inclusive victim cache. The CPU has an integrated memory controller with a built-in prefetcher.

Table 6.4: Hardware performance counters used for our experiments. We did not use all of the counters listed. Some of the counters have known errata. We gathered this list from PAPI [102] and the AMD and Intel reference manuals [10, 72].

stat	Phenom	Core 2	Pentium D
Retired Instructions	retired_instructions	instructions_retired	instr_completed:nbogus
Retired μops	retired_uops	uops_retired:any	uops_retired:nbogus
Elapsed Cycles	cpu_clk_unhalted	unhalted_core_cycles	global_power_events:running
L1 dCache Accesses	data_cache_accesses	l1d_all_ref	front_end_event:NBOGUS uops_type:TAGLOADS:TAGSTORES
L1 dCache Misses	data_cache_misses	l1d_pend_miss	n/a
L1 iCache References	instruction_cache_fetches	l1i_reads	uop_queue_writes:from_tc_build:from_tc_deliver
L1 iCache Misses	instruction_cache_misses	l1i_misses	bpu_fetch_request:tcmiss
L2 Cache References	data_cache_missses + instruction_cache_missses	l2_rqsts:self:any:mesi	bsq_cache_reference:rd_2ndL_miss:rd_2ndL_hits: rd_2ndL_hite:RD_2ndL_hitm
L2 Cache Misses	l2_cache_miss:data + l2_cache_miss:instructions	l2_lines_in:self:any	bsq_cache_reference:RD_2ndL_MISS
Branch Instructions	retired_branch_instructions	br_inst_exec	branch_retired:mmnp:mmnm:mmtp:mmtm
Branch Misses	retired_mispredicted_branch_instructions	br_missp_exec	branch_retired:mmnm:mmtm

6.4.4 Benchmarks

We use the SPEC CPU2000 [136] benchmarks for evaluation purposes, as they are long enough to provide interesting results, but at the same time short enough that the cycle-accurate results have a chance of finishing within a few weeks.

The benchmarks were compiled with `-O3 -msse3 -funroll-all-loops -ffast-math -static`.

The `vortex` benchmarks and some of the `perlbmk` benchmarks did not run; this is a limitation of the benchmarks themselves with modern compilers, and not an issue with our simulation methods. The same benchmarks fail on actual hardware.

We run full reference input sets for all experiments.

We ran all of the simulations on a large cluster of 3.4GHz Pentium D machines, identical to the system herein referred to as "Pentium D".

6.5 Absolute Results

We first investigate the absolute results returned by our various simulation methods. These are the results for one hardware configuration, without varying any of the simulation parameters.

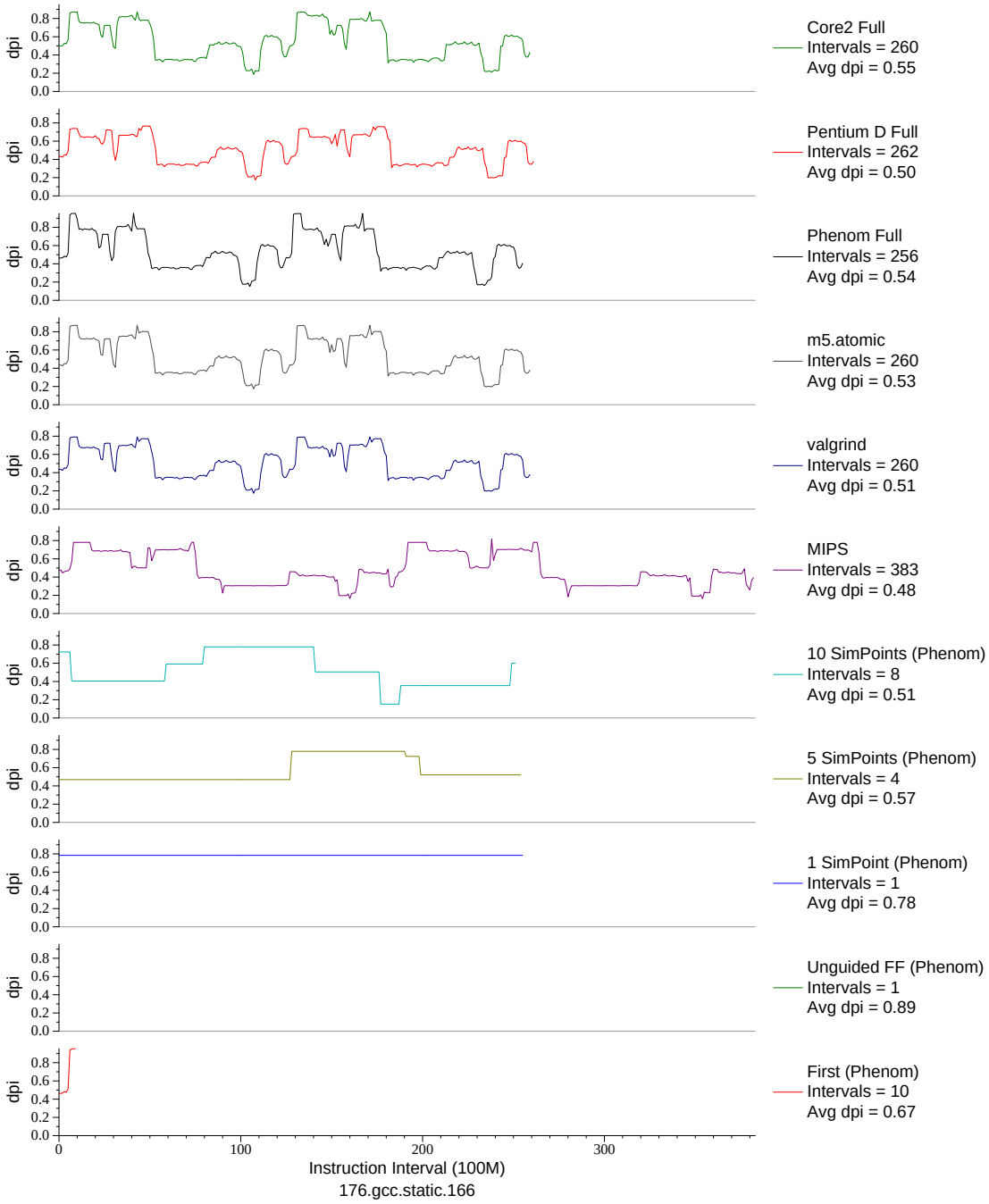

Figure 6.3: L1 data cache accesses per instruction. This plot shows that cache accesses per instruction is consistent across all actual machines, as well as the simulators. The MIPS results are very different. SimPoint results are shown for comparison

6.5.1 Phase Behavior Results

Figure 6.3 shows the phase behavior of L1 data cache accesses per instruction for `gcc.166` (full results for SPEC CPU2000 can be found in Appendix G). The three actual hardware implementations give practically identical plots for this metric, which is encouraging. Valgrind and m5 also give similar results. The MIPS results, while showing similar patterns, has many more instructions so any direct comparisons cannot be made. Also shown on the graph are the SimPoint results and the results of un-guided simulation.

6.5.2 L1 Instruction Cache

Figure 6.5 shows actual and predicted L1 instruction cache miss rates. Actual hardware measures icache references, while the DBI tools measure total instructions. In order to convert between the two, the average instruction size is needed. On RISC this is a fixed value, but x86 has variable-sized instructions. We scale the results based on an average number of bytes per instruction (shown in Figure 6.4). On our Phenom reference platform, a 16-byte load from icache is considered an instruction reference. The actual hardware does aggressive prefetching, always fetching the next block. The Valgrind rates are relatively close to actual hardware. m5 reports results much lower than real hardware, we have not yet determined the cause of this discrepancy.

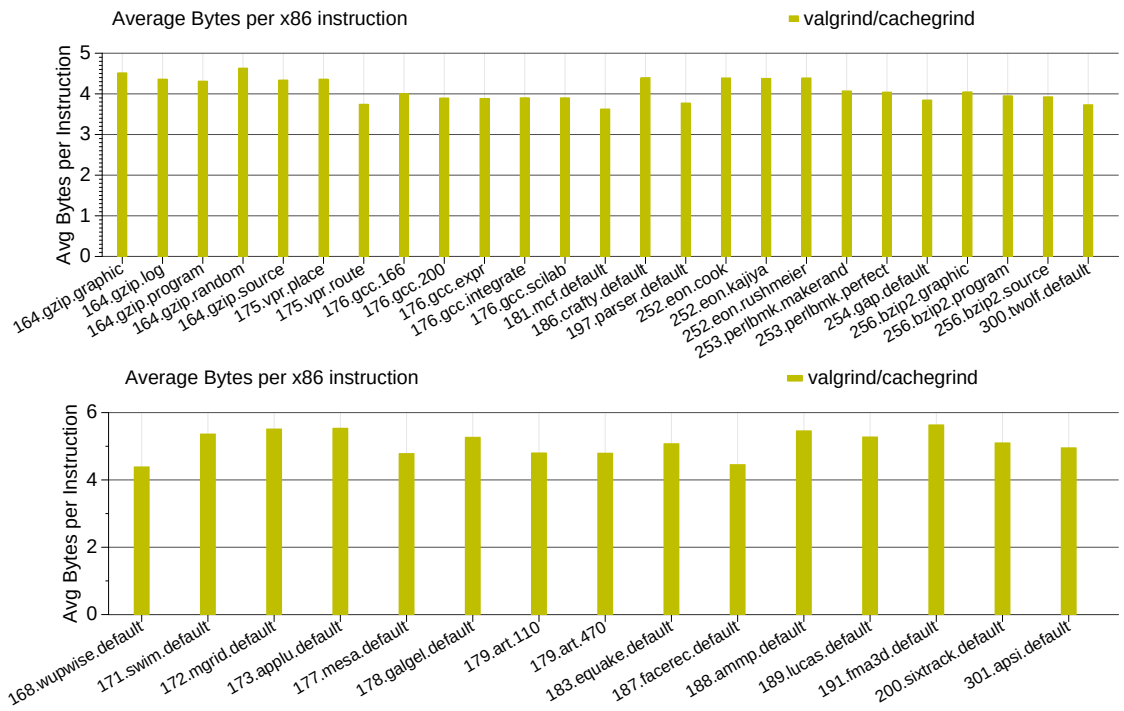

Figure 6.4: Average bytes per x86 instruction. For integer benchmarks the average is 4.0, for floating point it is 5.1. These values are needed when extrapolating cache miss rates when given only total retired instruction count.

6.5.3 Data Accesses per Thousand Instructions

Figure 6.6 shows data cache accesses per thousand instructions for the SPEC CPU2000 benchmarks. Most of the architectures show consistent results, and Valgrind and m5 are roughly the same. The one confusing point is the 32-bit result; the 32-bit binary generates many more cache accesses on the same machine and kernel than a 64-bit binary. This could be a compiler difference; it could also be the program having to split 64-bit memory accesses into two separate accesses.

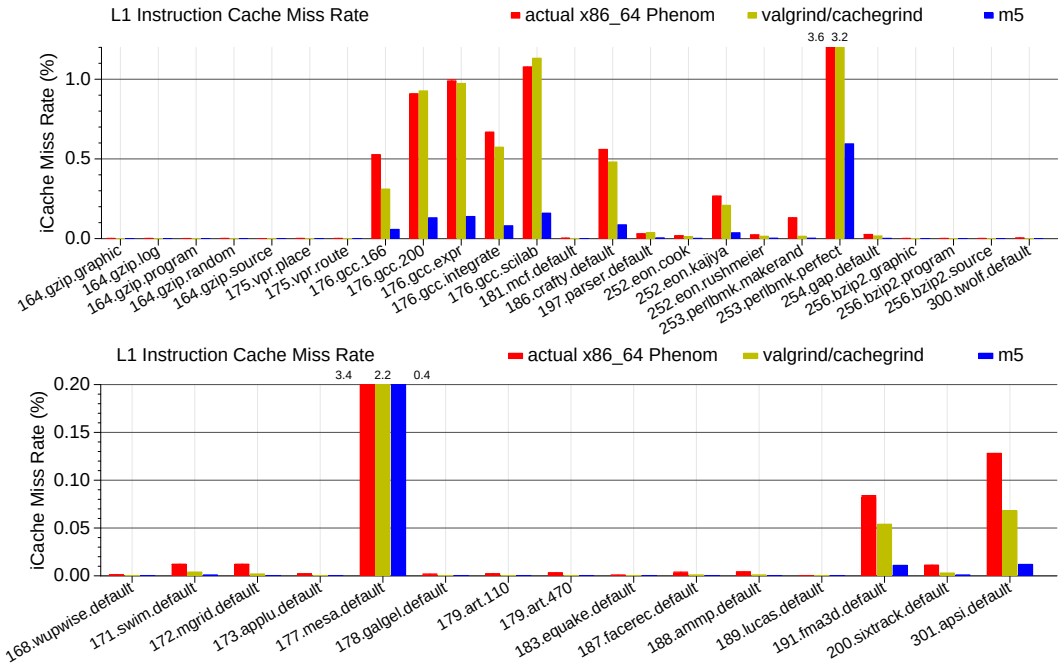

Figure 6.5: Instruction cache miss rate with integer benchmarks above and floating point below.

6.5.4 L1 Data Cache

Figure 6.7 shows L1 data cache miss rates for CPU2000. The `gcc` benchmarks have very high miss rates, in ways that the simulators do not expect. The rate is much higher than the miss rate when running the equivalent 32-bit binary. This is possibly due to the expansion to 64-bit pointers, as `gcc` is a pointer-heavy code. We conduct extra performance counter measurements that show the gcc benchmarks software prefetch more than the other benchmarks; this could be polluting the cache.

Figure 6.8 adds additional points to the previous graphs. All of the results presented are either simulating a Phenom-like cache or else running on an actual Phenom. 32-bit results are displayed, showing in detail that the `gcc` bench-

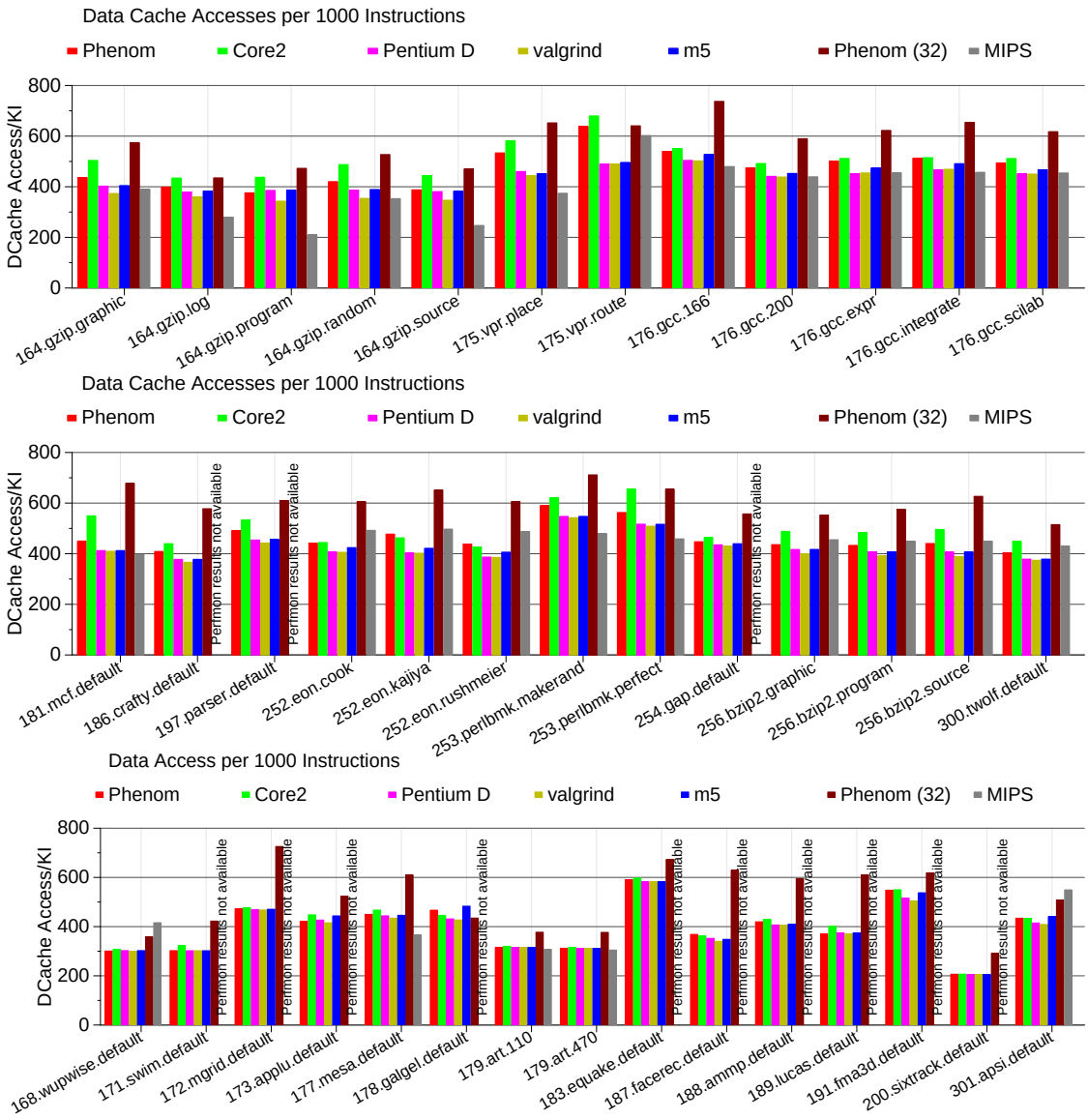

Figure 6.6: Data Accesses per Thousand Instructions for the SPEC CPU2000 benchmarks

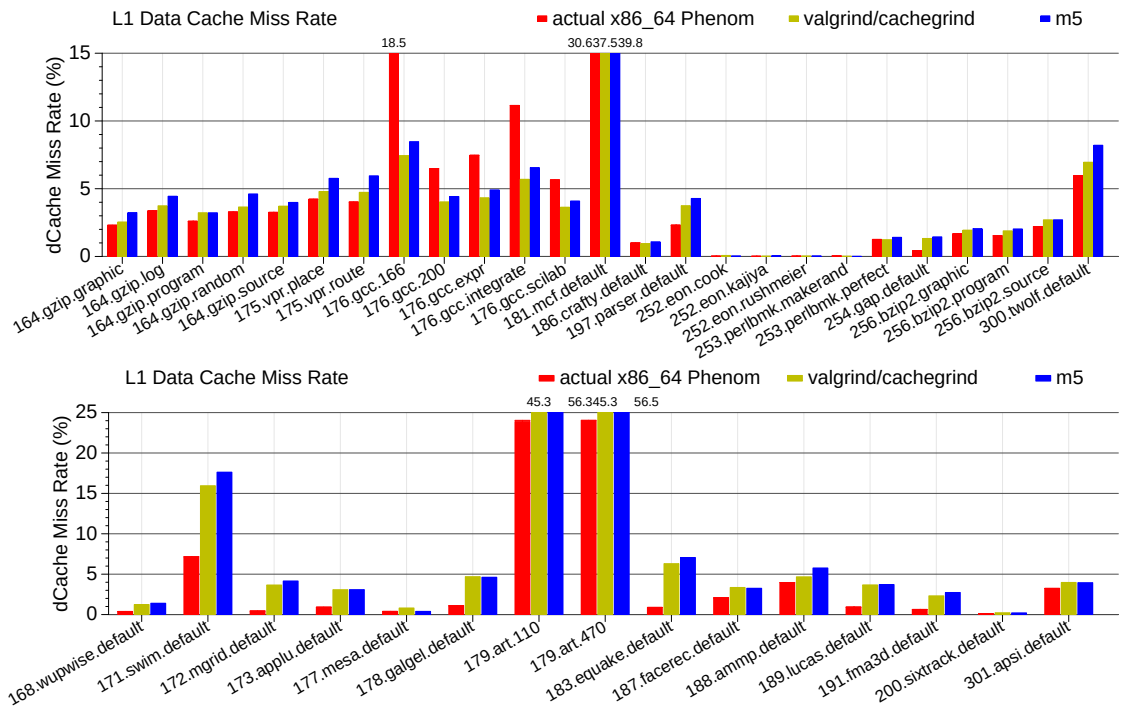

Figure 6.7: L1 data cache miss rate with integer benchmarks above and floating point below.

marks have vastly fewer cache misses than the equivalent 64-bit versions. The 32-bit floating point benchmark results are also different than 64-bit; this is possibly due to x87 versus SSE math differences. In most cases the simulators are overly pessimistic about the data cache rates. This is possibly because none of the simulators are modeling hardware prefetching, nor are they properly modeling the cache as exclusive. PPC results are shown too, on PPC Valgrind configured with the same cache parameters as x86 Valgrind. Those results are different, again showing that RISC traces cannot predict CISC results.

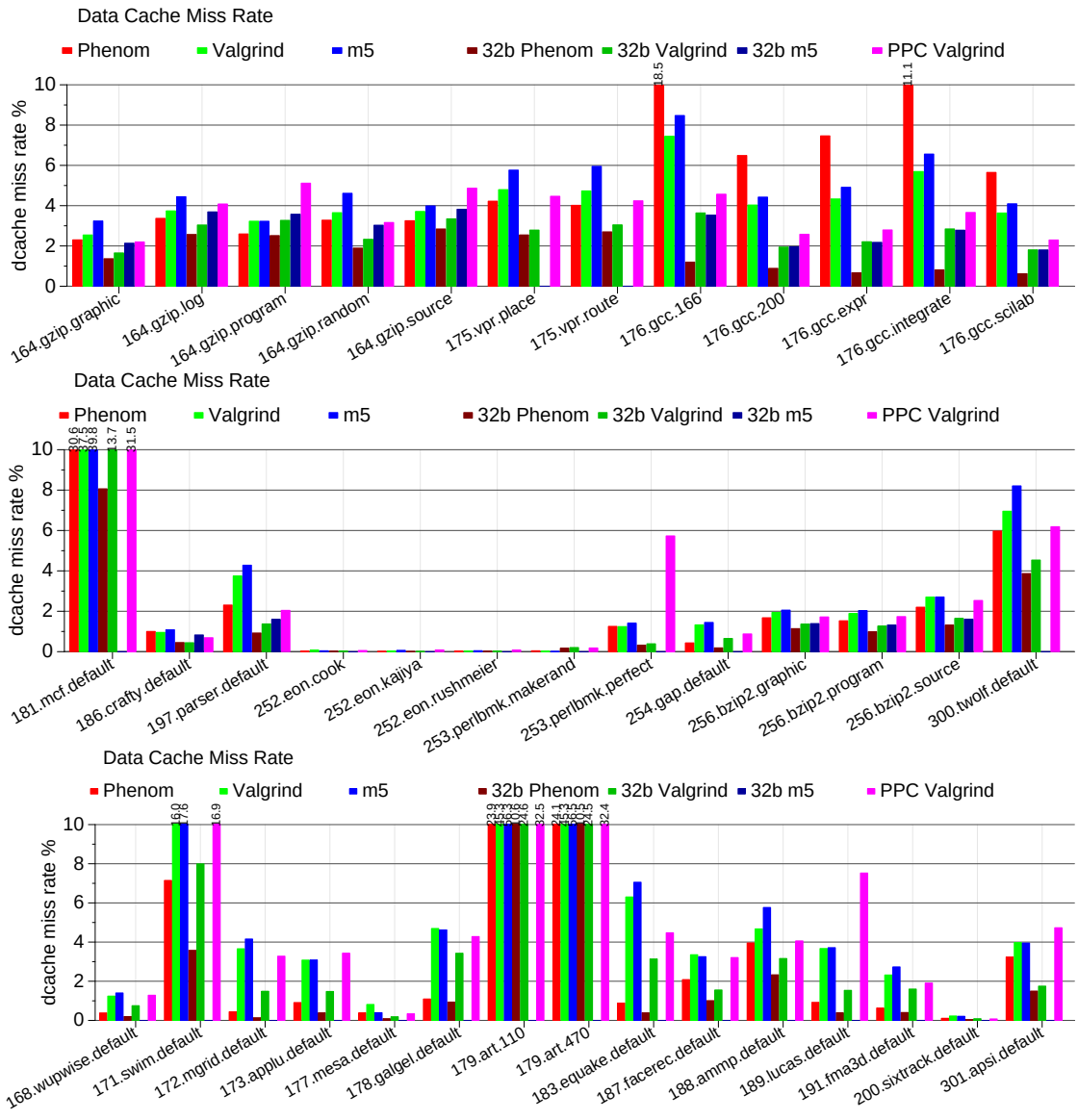

Figure 6.8: Dcache miss rates for Phenom-style cache

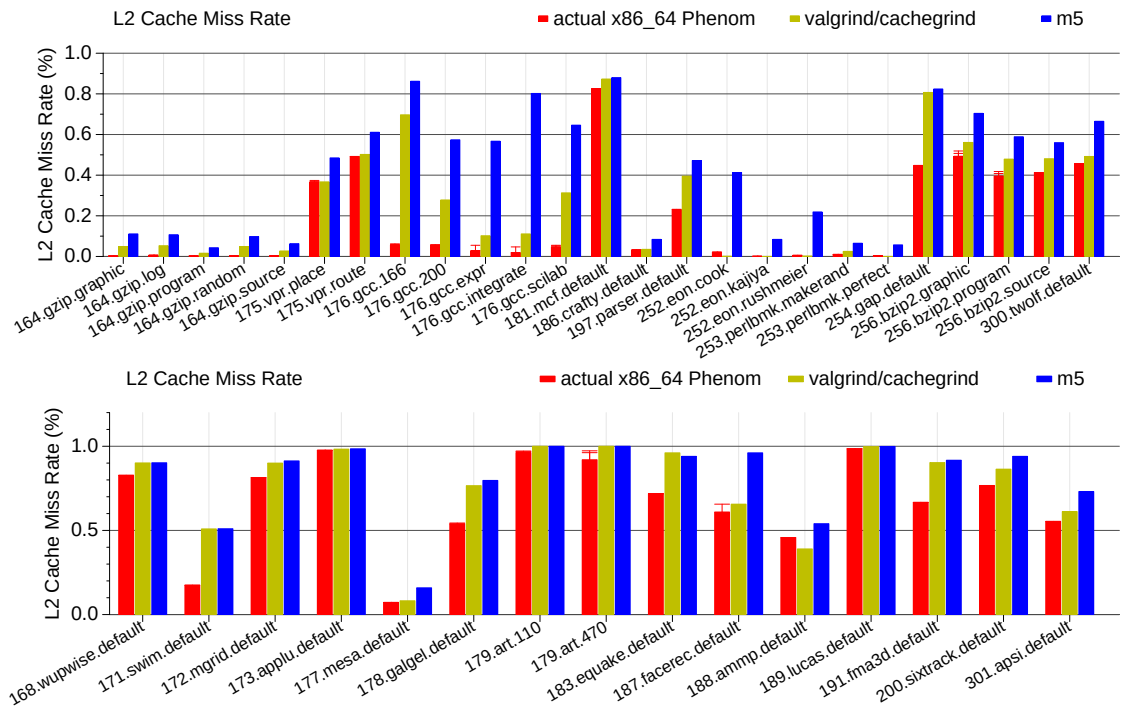

Figure 6.9: L2 cache miss rates, actual and simulated. The simulators are pessimistic; in the case of `gcc` severely so.

6.6 L2 Cache

Figure 6.9 shows L2 miss rates for CPU2000 on x86_64, both actual and simulated. The simulated results are pessimistic, severely so in the cases of `gcc` and `swim`. While large in relative terms, the absolute differences in the rates are relatively small. The benchmarks with the largest L2 error are also the ones that have large error with the L1 data cache, so this error might just be the L1 error propagating to the next level of the hierarchy.

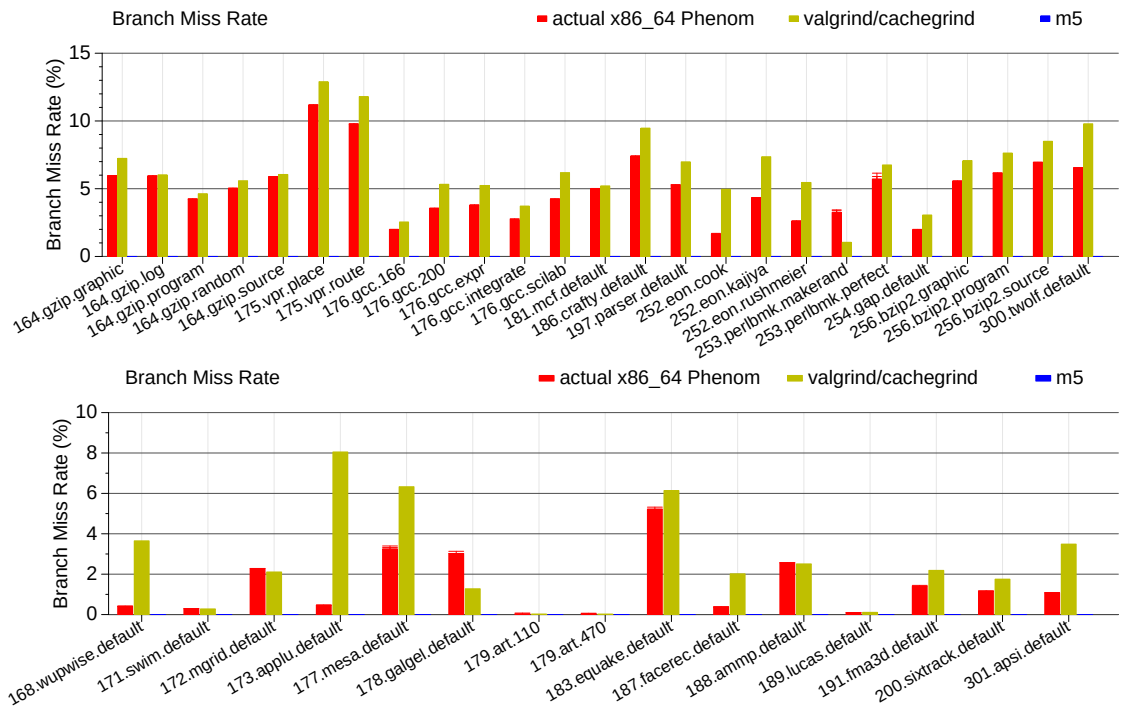

Figure 6.10: Branch predictor results for Valgrind and actual hardware. m5 currently cannot simulate branch prediction for x86_64

6.7 Branch Predictor

Figure 6.10 shows branch predictor results for Valgrind and actual hardware. m5 results are not shown, as m5 currently cannot simulate branch predictors on x86_64. The results match surprisingly well for the integer codes, considering Valgrind is modeling a simplistic 16k 2-bit up/down counter predictor. The results are not as good for floating point results, which is a bit surprising as typically floating point branches should be easier to predict. This could mean that the Phenom has a predictor specially optimized for floating point codes.

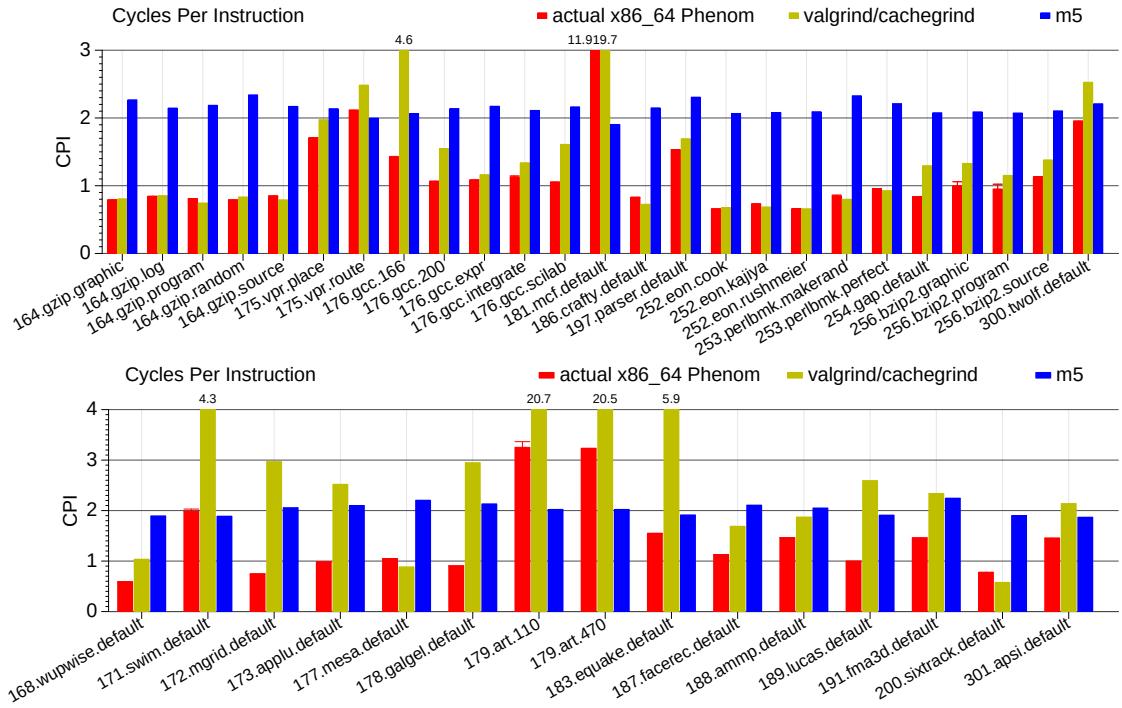

Figure 6.11: CPI results with integer above and floating point below. Valgrind cycle times are estimated based on cache and branch predictor behavior.

6.8 CPI

Figure 6.11 shows CPI results. The Valgrind cycle counts are estimated, based on a formula similar to the one in Section 5.5.1. The Valgrind results are impressively good for integer benchmarks, though they are off for `gcc` (which is unsurprising as the data cache results for `gcc` are poor, skewing the cycle estimate). The Valgrind floating point results are poor, possibly due to the lack of good branch prediction results. The m5 results are poor overall, as it is simulating a simple atomic CPU where only one instruction finishes at a time. Since it lacks any super-scalar simulation at all, the cycles are always going to be off compared to an out-of-order processor.

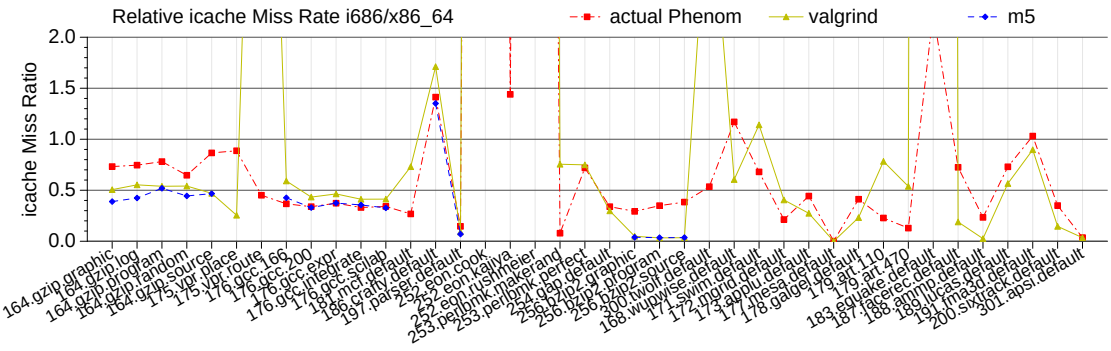

Figure 6.12: Relative instruction cache miss rate ratios when moving from 32-bit to 64-bit

6.9 Relative Results

As with the RISC results, we present results that compare how the various methods predict improvement when changing an architectural feature. Unlike the RISC case, none of the chips we have support changing architectural features on the fly. Instead, we compare results when moving from 32-bit to 64-bit on the same machine. Real hardware, Valgrind, and m5 all support running both 32-bit and 64-bit x86 binaries. Unfortunately m5 cannot run the full CPU2000 benchmarks in 32-bit mode due primarily to unimplemented x87 floating point support. This severely limits the number of benchmarks that can be compared using m5.

6.9.1 L1 Instruction Cache

Figure 6.12 shows relative icache results when moving from 32-bit to 64-bit. While in general the simulated results track actual performance counter data,

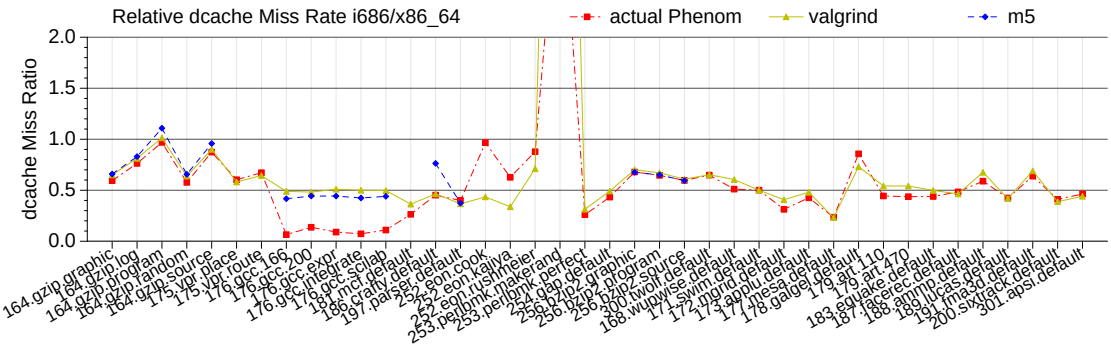

Figure 6.13: Relative L1 data cache miss rate ratios when moving from 32-bit to 64-bit

the ratio is not as close as it could be. This might have to do with actual hardware engaging in hardware prefetching.

6.9.2 L1 Data Cache

Figure 6.13 shows relative L1 data cache miss rate ratios when moving from 32-bit to 64-bit. These results track much better than the instruction cache results. The primary outliers seem to be the `gcc` benchmarks (discussed earlier) and the `eon` benchmarks.

6.9.3 L2 Cache

Figure 6.14 shows relative L2 miss rate ratios. Unfortunately the integer results are not good. Valgrind and m5 cannot predict `gzip` or `gcc` behavior; this is possible because those benchmark's actual performance is dramatically worse when moving to 64-bit. Valgrind does a much better job at predicting floating

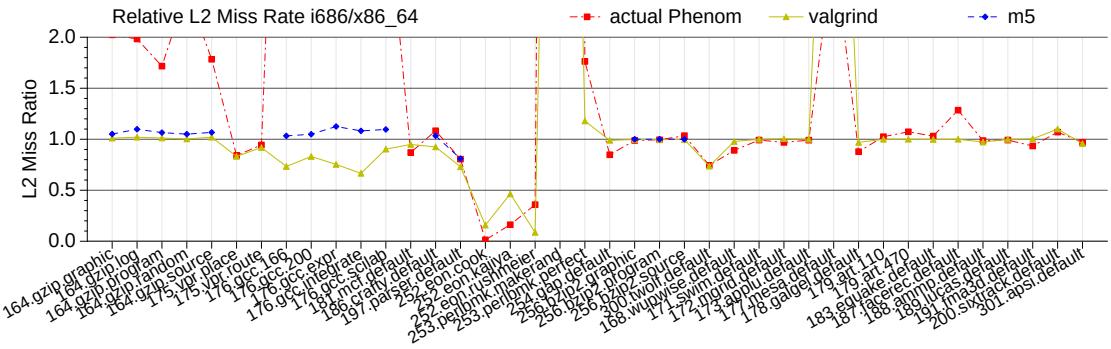

Figure 6.14: Relative L1 data cache miss rate ratios when moving from 32-bit to 64-bit

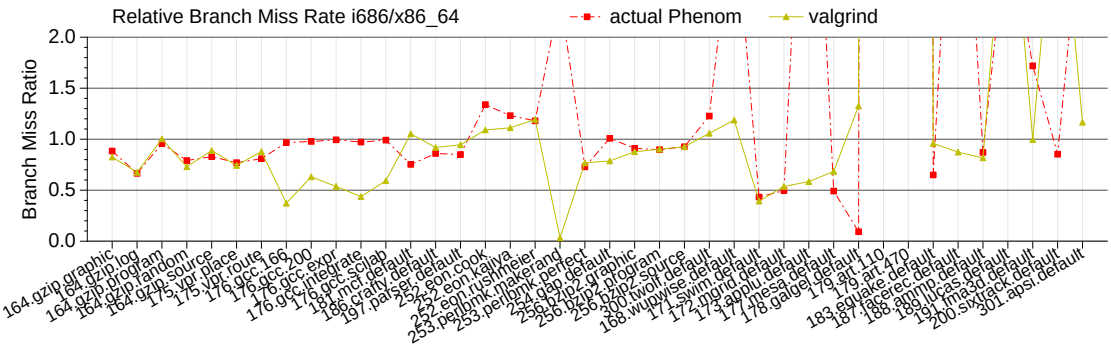

Figure 6.15: Relative branch predictor miss rate ratios when moving from 32-bit to 64-bit

point behavior.

6.9.4 Branch Predictor

Figure 6.15 shows the relative change in branch predictor results when moving from 32-bit to 64-bit. m5 is not represented, as currently it lacks branch predictor support for x86 and x86_64. Unfortunately it turns out that Valgrind only prop-

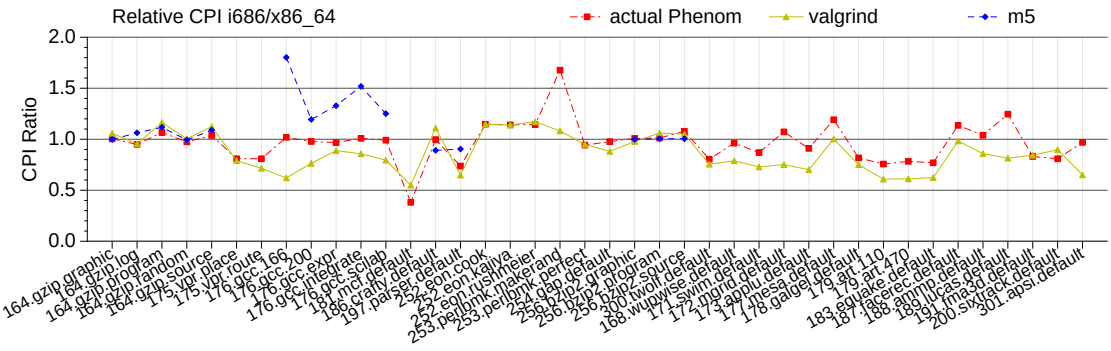

Figure 6.16: Relative CPI ratios when moving from 32-bit to 64-bit

erly predicts a subset of the integer benchmarks. One unexpected data point is the wildly different branch predictor accuracy when moving to 64-bit on real hardware.

6.9.5 CPI

Figure 6.16 shows relative CPI results. Unfortunately Valgrind does a poor job of predicting, although this is not surprising as Valgrind's cycle count is only an estimate. m5 is even worse, but it has its own issues with cycle count, as described in the absolute results section. Without major changes to the simulator, it is not possible to use these tools to predict an architectural change of this magnitude.

6.10 Summary

Unlike the RISC results found in Chapter 5, we find that current tools and simulators are not up to the task of predicting performance on CISC systems. It is possible that a more faithful model of the underlying architectures would generate better results. It is also possible that the 32-bit to 64-bit comparison has too many variables; the RISC branch-prediction study might have been an easier target for this type of analysis. This area of research could use further study.

CHAPTER 7

MULTI-CORE VALIDATION CONCERNS

Chip multi-processing (CMP) systems retain all the validation concerns found with single-core systems (as described in Chapter 4) while adding new and more complex issues. We briefly address the issues encountered when extending our simulation methodology work to handle multiple cores.

7.1 Performance Counters

Most CMP systems support per-core performance counter measurements. There are some counter issues that do not occur on single core machines; for example most CMP systems have some number of resources that are shared between the cores, such as L3 caches or memory controllers. When measuring statistics for these structures, it can be unclear which core owns these counts. These troublesome shared resources are sometimes referred to as the "uncore" and the perfmon2 tool makes it possible to count these. Unfortunately this often involves extra work, or else forces counts to be taken system-wide even if the thread of interest is only running on one of the cores.

7.2 Deterministic Execution

The problem of deterministic execution becomes even more pronounced once more cores are added to a system. The theoretical rock of stability in our previous analysis, the retired instruction count, no longer has any guarantees. Once multiple threads are running, most hope of deterministic execution are lost. The

Operating System takes on a larger role, as scheduling decisions by the OS can vastly change overall system performance.

When performing validation, is is important that the simulator is running the same exact code as the real hardware. This is much harder on CMP systems. Many cycle-accurate simulators do not even model the Operating System at all, and even if they did, synching the scheduling decisions between a simulator and actual hardware is not trivial.

If execution cannot be made deterministic, then comparisons between simulation and hardware are meaningless.

There has been a lot of work toward deterministic multi-thread execution (see Section 2.9). Unfortunately many of the implementations are at the hardware level and thus require low-level architectural changes. Some recent examples of such solutions are Capo [98], DMP [44], Delorean [97] and Flight Data Recorder [165].

An ideal deterministic execution method for validation work would be software-only, require limited changes to the executables being run, and should work unmodified on both real hardware and in a simulator. The recent Kendo [114] project meets all of these criteria. Kendo uses hardware performance counters to enforce deterministic context switching via a modified version of the pthreads library. The retired_stores performance counter is used as a reference count as they (like us) found that other counters like (retired_instructions) include interrupt counts and other undesirable noise. Using Kendo adds an average overhead of 16% to execution time, which is unfortunate, but worth the sacrifice.

For our validation work we would have liked to use Kendo, but unfortunately despite originally saying it would be available for download, at the time of writing this the authors were still not ready to release it. Thus our validation attempts were made without the use of CMP determinism, with all the problems involved therein.

CHAPTER 8

MULTI-CORE RESULTS

Our original plan was to generate multi-core results using Valgrind/Ruby, m5, and actual hardware performance counters and then compare the results. This would have been a natural extension to the RISC results in Chapter 5 and the single-core CISC results in Chapter 6.

Unfortunately the standalone Ruby CMP cache simulator was not mature enough to do this type of research. The m5 simulator's x86_64 support was also not ready for this type of experiment. Nor were other x86_64 simulators such as PTLSim.

We investigated maybe using other architectures, but were limited by the hardware we had access to that performance counters were fully working. This eliminated Alpha, MIPS and SPARC.

In then end, what we present are some preliminary results showing that we get sane memory access patterns across DBI, cycle-accurate and real hardware. However the actual end results of the cache simulations cannot be compared.

8.1 Methodology

We run experiments using some of the SPEC OMP [137] benchmarks. We compile with the Intel ICC compiler, as the benchmarks for some reason do not scale when compiled with gcc 4.4.

8.1.1 Performance Counters

We run our tests on a 4-core AMD Phenom system running the 2.6.29 kernel patched to enable perfmon2 [51] performance counter support.

The Phenom system has rich performance counter support, allowing counts on a per-core basis. Detailed overall system counts are available for shared resources, such as the L3 caches and the memory controller.

8.1.2 DBI Simulation

Various DBI tools have support for CMP simulation. For user-space only tools, this involves intercepting the various thread and process creation system calls and handling the situation appropriately. Some DBI tools, such as Valgrind, handle the multi-thread case but can only themselves run one thread at a time. This in effect serializes the multi-thread execution. Despite this serialization, CMP results can still be effectively used if the traces generated have enough information to re-create the parallel execution. Running in a serial fashion though does cause a linear slowdown in execution for each additional thread being run.

Not all DBI tools force serialization on multi-thread executions. The Pin tool is capable of spawning a separate DBI instance for each thread, allowing a program to be simulated in a manner much closer to native execution [65]. This could lead to faster trace generation than with Valgrind.

For collecting DBI CMP memory traces we use Valgrind 3.5 with a custom tool (based on our exp-bbv tool) that generates memory and instruction traces. These traces are fed into an external program via a named pipe that counts and

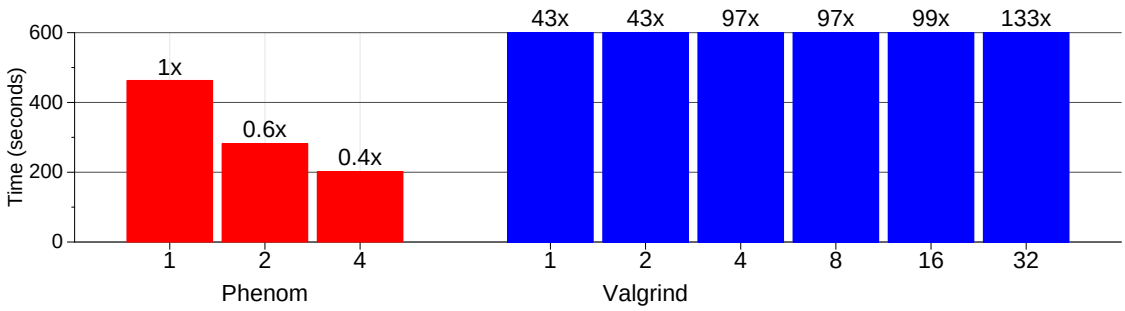

Figure 8.1: `equake_m` run times for varying number of threads, both on actual hardware and Valgrind

analyzes the references.

8.1.3 Cycle-accurate Simulation

We had hoped to use m5 for x86_64 multi-threaded cache simulation. However the user-mode support for multi-core is not working, and nor is the full-system mode that would allow running multithreaded benchmarks on top of a full simulated operating system. In the end we did not conduct any cycle-accurate multicore simulations.

8.2 Results

Figure 8.1 shows run times when running `equake_m` on real hardware and on Valgrind. On real hardware the benchmark scales with number of CPUs, although not purely linearly. Valgrind has interesting behavior; one would expect

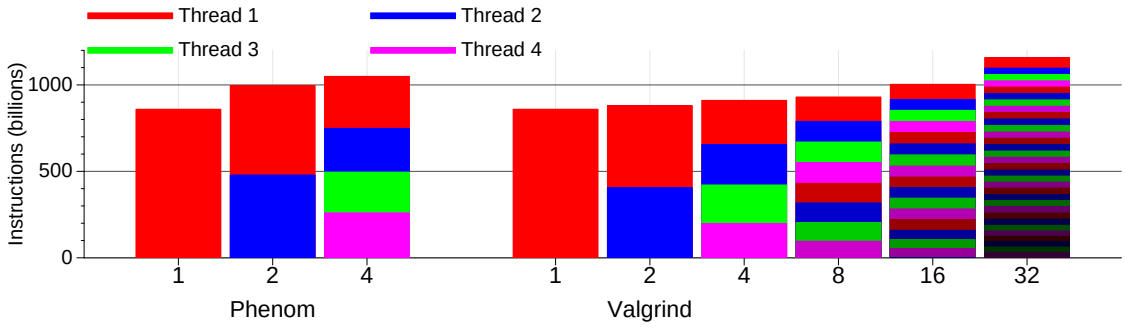

Figure 8.2: `equake_m` retired instruction counts for varying number of threads, both on real hardware and Valgrind

the total run time to stay approximately the same, as for the benchmark the same amount of total work is being done, it is just being split between cores. For the one and two thread cases this holds, but adding additional threads dramatically increases the run times. This could be due to an artifact with Valgrind's internal thread scheduling mechanism, possibly conflicting with the way the OpenMP library distributes the work among threads.

Figure 8.2 shows per-thread retired instruction counts for `equake_m` on real hardware and on Valgrind. In each case there is a helper scheduler thread running in addition to the shown threads, but it is proportionally so few instructions it is not visible on the graph. The overall retired instruction counts grow as threads are added due to multi-threading overheads. This overhead is higher on real hardware, due to locking overheads from concurrent execution that do not occur under the Valgrind DBI tool. It is encouraging that the relative ratio of instructions per thread is consistent between real hardware and simulation. The Valgrind tool allows running experiments on more threads than the actual hardware has available; this allows running experiments for machines with more

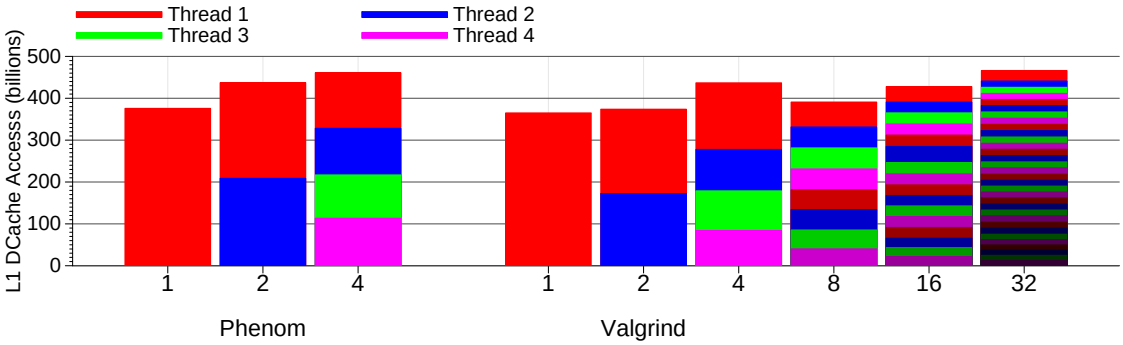

Figure 8.3: `equake_m` L1 dcache access counts for varying number of threads, both on real hardware and Valgrind

cores than currently available.

Figure 8.3 shows per-thread L1 DCache accesses when running `equake_m` on real hardware and Valgrind. As before, there is an additional helper thread too small to be visible on the plots. It is encouraging that the relative ratios of memory accesses per thread is similar between hardware and Valgrind. Especially note that thread one has proportionately more accesses in both instances. We cannot explain why the total number of memory accesses drops when moving to 8 threads on Valgrind. We do not have 8-core hardware so we do not know if the same thing happens on an actual machine. We find it encouraging that the cache results for Valgrind match so closely, as it gives hope that once a multi-threaded cache simulator is available that with proper tuning it can produce outputs just as good as its inputs.

8.3 Summary

Even though we are not able to generate the results from CMP cache simulation on x86, we have conducted preliminary experiments that show that the data cache accesses that would be fed into the simulator are sane and match real hardware. This gives hope that once a CMP simulator becomes available, that methodology similar to our single-core methodology could be used for validation purposes.

CHAPTER 9

CONCLUSION AND FUTURE WORK

Our goal is to speed simulation times of architectural simulations without affecting accuracy. First we investigate reduced execution methods, concentrating on the SimPoint methodology. We find that SimPoint has much higher accuracy than other commonly used methods, but it can still take long run times when attempting to generate high accuracy results. We next look at using DBI tools, which run orders of magnitude faster than cycle-accurate simulation, to generate results using full input sets. We find that it is simple to get good results using DBI means on RISC platforms. Unfortunately we find it is not as simple to get good results on more modern CISC machines. We begin preliminary investigations of whether the DBI method of simulation would work on CMP systems, as opposed to single core machines previously investigated.

9.1 Results Summary

Simulation time is of critical importance to most computer architects. Many are willing to trade accuracy by any means necessary so that their experiments can finish in a reasonable amount of time.

Figure 9.1 shows speed versus accuracy tradeoffs for the various simulation methods that we investigate. The results are for the CPU2000 benchmarks. Not all of these are actual results; approximations were made where DBI simulation is 376x slower than native, function simulation is 390x slower than native, and cycle-accurate simulation is 3900x slower than native (these values match the ones found with Qemu and SESC in Chapter 5). The results also assume per-

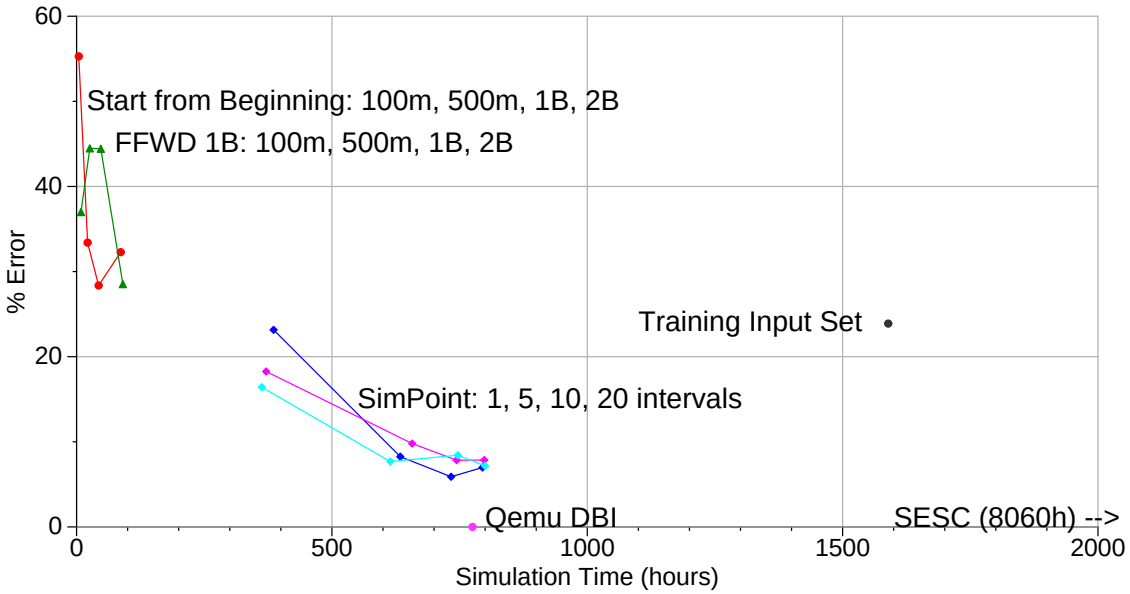

Figure 9.1: Speed vs Accuracy tradeoffs of the various simulation methods on SPEC CPU2000, assuming perfect simulation

fect results, that is the simulators generate the same results that performance counters would. Actual error rates will be worse, accumulating error from the simulator. The results show that for accuracy, nothing can beat DBI. Full inputs can be run in the time it takes to run 20 SimPoints in the cycle-accurate simulator. Assuming that the same accuracy can be obtained with DBI as with cycle-accurate, using DBI is almost always the winner. There are other simulation methods that might also compare favorably; the SimPoint results assume functional fast-forwarding for each run. If a method such as SimSnap [143] is used to leverage snapshots, so that fast forwarding is instantaneous, then the slowdown times would be reduced even more.

The best possible comparison would involve having the full SimPoint measurements and accuracy including simulator overhead for both cycle-accurate and DBI, but unfortunately we did not have time to generate those results.

9.2 Future Work

The most important future work is the completion of the CMP work started in Chapter 8. The various projects involved, most notably m5 and gem5/ruby are under heavy development and may become usable at any time. Barring that, PTLsim also may gain full CMP support and be ready for the experiments we need. Once that happens, there is hope that DBI methods can be validated against both hardware and cycle accurate simulators on CMP systems.

Another future work is to make use of the faster execution times enabled by DBI-based simulation. One major use would be modeling DRAM systems in full detail, possibly by using DRAMsim [150]. The main problem holding back detailed DRAM simulation is slow simulation time, something that is addressed by or DBI simulation methods.

9.3 Conclusion

DBI-based methods make the best of the speed versus accuracy tradeoff in computer architectural simulation. We encourage researchers to use DBI methods if possible, to allow running longer-running more complete simulations, including simulations of overlooked (due to speed) subsystems, such as DRAM and I/O. Modern systems continue to grow in complexity, and without moving to faster methodologies, such as DBI, we will rapidly lose the ability to have any confidence in simulation results.

APPENDIX A

THE LOST ART OF ASSEMBLY LANGUAGE PROGRAMMING

When debugging simulators and DBI tools, being well versed in various assembly languages helps immensely. Assembly is optimal for designing small test cases, especially ones where the simulator is having errors before getting past the C library. Obscure bugs and reproducible test cases for external distribution are also best done in assembly.

Once you are well versed in writing tiny assembly language, all the tools are available to explore the nature of code density on modern processors.

A.1 Benefits of Code Density

Dense code yields many benefits. The L1 instruction cache can hold more instructions, which usually results in fewer cache misses [139]. Less bandwidth is required to fetch instructions from memory and disk [38], and less storage is needed to hold program images. With fewer instructions, more data fits in a combined L2 cache. Also, on modern multi-threaded processors, multiple threads share limited L1 cache space, so having fewer instructions can be advantageous. Denser code causes fewer TLB misses, since the code requires fewer virtual memory pages. Modern Intel processors, for instance, can execute compact loops entirely from the instruction buffer, removing the need for L1 I-cache accesses. Finally, the ability to consistently generate denser code can conserve power, since it enables smaller microarchitectural structures and uses less bandwidth [63, 149, 177, 19, 15].

Obviously, these benefits can come at a cost. For example, a denser ISA

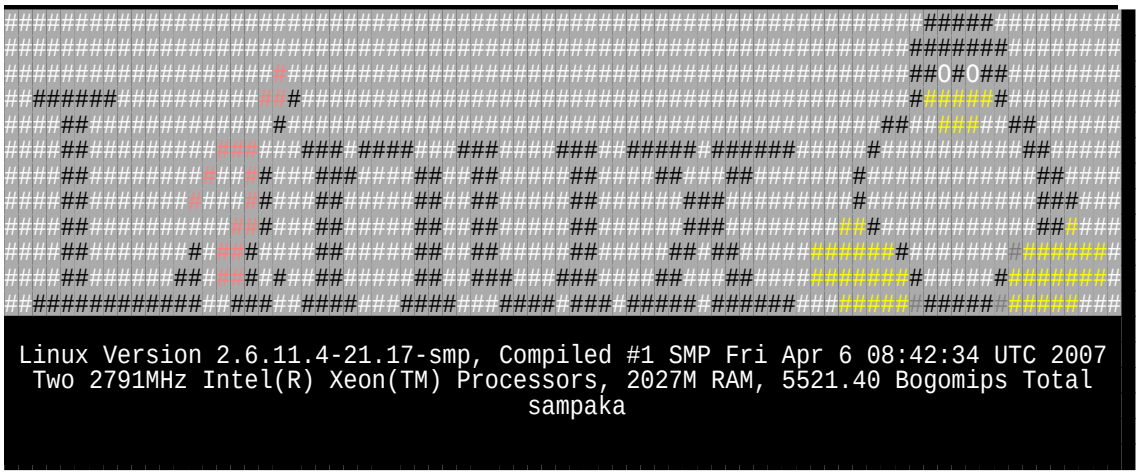

Figure A.1: Sample output from the `linux_logo` benchmark

might require larger (and thus slower) pipeline decode stages, more complicated compilers, smaller logical register set sizes (due to limitations in the number of bits available in instructions), or even slower and more complex functional units. Compilers tend to optimize for performance, not size (even though the two are inextricably related): obtaining optimal code density often requires hand-tuned assembly language, which represents yet another tradeoff in terms of programmer time and maintainability. The current push for using CISC chips in the embedded market [133] forces a re-evaluation of existing ISAs.

A.2 Methodology

Investigations of code density often use microbenchmarks (which tend to be short and not representative of actual workloads) or else industry standard benchmarks (which are written in high-level languages and thus are limited by compiler code generation capabilities). As a compromise, we take an actual system utility, but convert it into pure assembly language in order to directly

Table A.1: Summary of investigated architectures

Type	arch	endian*	bits	instr len (bytes)	op args	GP int regs	unaligned ld/st	auto-inc address	hw div	stat flags	branch delay	predi-cation
VLIW	IA64	little	64	16/3†	3	127,zero	no	yes	no	yes	no	yes
RISC	Alpha	little	64	4	3	31, zero	no	no	no	no	no	no
	ARM	little	32	4	3	15,PC	no	yes	no	yes	no	yes
	m88k	big	32	4	3	31,zero	no	no	Q only	no	optional	no
	MicroBlaze	big	32	4	3	31,zero	no	no	Q only**	no	optional	no
	MIPS	big	32/64	4	3	31,hi/lo,zero	yes**	no	yes	no	yes	no
	PA-RISC	big	32/64	4	3	31,zero	no	no	part	no	yes	no
	PPC	big	32/64	4	3	32	yes	yes	Q only	yes	no	no
	SPARC	big	32/64	4	3	63-527,zero‡	no	no	Q only	yes	yes	no
CISC	m68k	big	32	2-22	2	16	yes	yes	yes	yes	no	no
	s390	big	32/64	2-6	2	16	yes	no	yes	yes	no	no
	VAX	big	32	1-54	3	16	yes	yes	yes	yes	no	no
	x86	little	32	1-15	2	8	yes	yes	yes	yes	no	no
	x86_64	little	32/64	1-15	2	16	yes	yes	yes	yes	no	no
Embedded	AVR32	big	32	2	2	15,PC	yes	yes	yes	yes	no	no
	CRISv32	little	32	2-6	2	16,zero,special	yes	yes	part	yes	yes	no
	SH3	little	32	2	2	16,MAC	no	yes	part	yes	yes	no
	THUMB	little	32	2	2	8/15,PC	no	yes	no	yes	no	no
8/16-bit	6502	little	8	1-3	1	3	yes	no	no	yes	no	no
	PDP-11	little	16	2-6	2	6,sp,pc	no	yes	yes**	yes	no	no
	z80	little	8	1-4	2	18	no	lim	no	yes	no	no

* on the machine we used † 16-byte bundle has 3 instructions ‡ register windows, only 32 visible ** many implementations

interact with the underlying ISA. We hand-optimize it for size, attempting to create the smallest binary possible, even if this potentially creates slower code. The program we choose, `linux_logo` [151], is a utility available with many Linux distributions. When given a sufficiently large input set, its characteristics are similar to the `stringsearch` benchmark included with the MiBench [60] suite. The program executes various syscalls to gather system information, then displays this info along with a colorful ASCII penguin (Figure A.1 shows sample output).

The stock `linux_logo` program contains a multitude of features and command line options; we remove all but the minimum for simplicity. Remaining code is divided into two parts: the first decodes and displays the text logo, which is packed using LZSS compression [176, 140]; the second prints system information, which is gathered by reading the Linux `/proc/cpuinfo` file, in addition to invoking the `uname()` and `sysinfo()` syscalls. Major subroutines include string copying, string searching, integer to ASCII conversion, and centering routines. The code makes system calls directly to avoid C library overheads. Code is assembled with the GNU assembler and is linked with GNU ld. Executables are stripped of non-essential data using the `sstrip` "super strip" program [124], an enhanced version of the UNIX strip command. Executables are tested on actual hardware or under an emulator where hardware is unavailable.

We attempt to optimize each architecture's code to the minimum possible size without corrupting correct results. For RISC architectures with fixed-length instructions this is easier: typically, there is only one way to express an operation, so there are limitations to clever implementations. Optimizations are lim-

ited to trying to load 32-bit constants in a small area, using registers instead of memory, and using tail merging to shorten procedure lengths. CISC architectures provide many more opportunities to decrease code size, but it is much more difficult to track optimizations due to variable-length instructions. Optimizing for density requires frequent disassembler checks to verify sizes of individual instructions. Interestingly, we find that the "do-everything" super-CISC instructions available on these systems can often be implemented with a smaller set of simpler CISC instructions.

A.3 Architectural Notes

Table A.1 lists relevant features of the architectures of interest. We present a broad overview of these architectures.

VLIW: Very Long Instruction Word (VLIW) architectures are designed to take advantage of parallelism in code. If the code is not inherently parallel (and ours is not), code density suffers, and many operations are wasted as nops. Writing compact VLIW code can be hard: resolving dependences correctly is a difficult task for compilers, and an even more difficult task for programmers writing assembly by hand. VLIW can be designed with code density in mind: e.g., the WM [161] architecture could exploit two operations per instruction in over two-thirds of all cases. The only VLIW architecture we investigate is Intel's IA64 [71].

RISC: Reduced Instruction Set Computers (RISC) emphasize simple architectures with easy to decode instructions. Instruction length is fixed at four bytes, which necessitates inefficiency in instruction encoding. These are load-

store architectures, which require moving memory values into registers before operating on them (this negatively impacts code density). Some of these architectures stretch the definition of "reduced"; the PowerPC architecture has nine different add instructions, and has the `rlwimi` (rotate left word immediate then mask insert) instruction, which takes five parameters. We investigate the Alpha [36], ARM [11], m88k [100], MicroBlaze [164], MIPS [96], PA-RISC [67], PowerPC [70], and SPARC [142] ISAs.

CISC: Complex Instruction Set Computers (CISC) tend to have high code density. Most CISC architectures have variable-sized instructions, which makes processor decode more complicated, but allows for dense code. An example of a dense "complex" instruction is the x86 one-byte `lodsb` instruction, which both loads a byte from memory and increments a pointer. Another impressively complex instruction is the VAX `matchc`, which does a full "find substring x inside of string y in memory." Compilers often have difficulty using these instructions appropriately, so this potential for density can be wasted. Also, these instructions may not be shorter or faster than a set of simpler instructions performing the same operations. We investigate the m68k [101], s390 [69], VAX [46], x86 [73], and AMD64 [7] ISAs.

Embedded: Modern advances in CPU design have pushed the limits of what qualifies as "embedded". We use the term to refer to any architecture with a fixed two-byte instruction length, but capable of running a modern 32-bit Linux kernel. These processors tend to have consistently small code sizes, but can still be beaten by variable-instruction length CISC systems. We investigate the AVR32 [12], CRISv32 [14], SH3 [126], and ARM THUMB [11] ISAs.

Table A.2: Correlations of architectural features to binary size

Correlation Coefficient	Architectural Parameter
0.9381	Minimum possible instruction length
0.9116	Number of integer registers
0.7823	Virtual address of first instruction
0.6607	Architecture has a zero register
0.6159	Bit-width
0.4982	Number of operands in each instruction
0.3129	Year the architecture was introduced
-0.0021	Branch delay slot
-0.0809	Machine is big-endian
-0.2121	Auto-incrementing addressing scheme
-0.2521	Hardware status flags (zero/overflow/etc.)
-0.3653	Unaligned load/store available
-0.3854	Hardware divide in ALU

8 and 16 bit: For comparison purposes we investigate older processors with smaller word sizes. Such CPUs are still used for embedded systems, and they are designed for use where code density is a much more critical concern. We investigate the 6502 [99], PDP-11 [45], and z80 [175] ISAs.

A.4 Code Density Findings

Table A.2 shows how architectural features contribute to code size. A positive correlation means that high values of the feature increase code size; a negative correlation means that high values decrease code size. Figure A.2 shows total binary sizes across the investigated architectures and Figures A.3, A.4, A.5, and A.6 show code sizes of various components. We detail causes of these trends below.

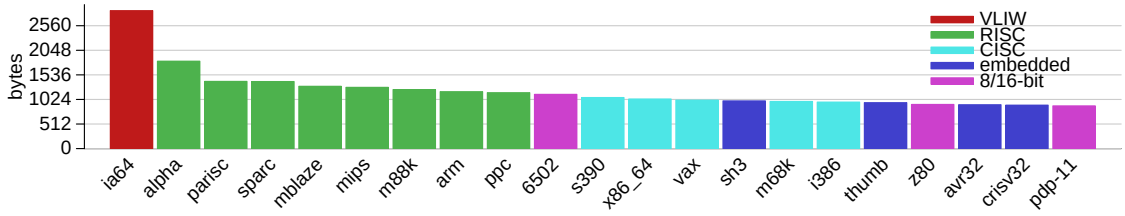

Figure A.2: Total size of benchmarks (includes some platform-specific code, so does not strictly reflect code density)

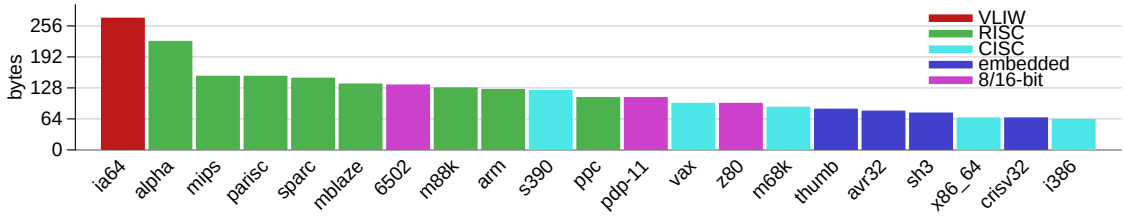

Figure A.3: Size of LZSS decompression code

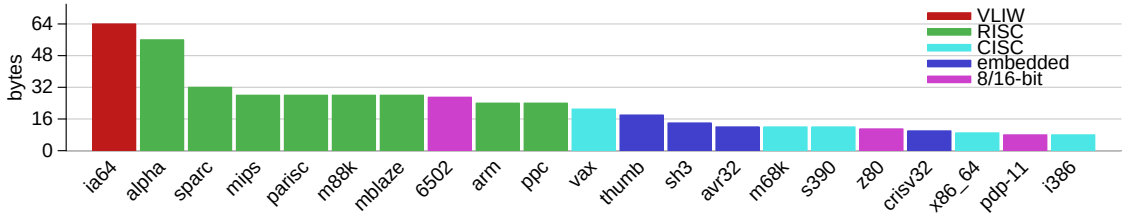

Figure A.4: Size of string concatenation code (machines with auto-increment addressing modes and dedicated string instructions perform better)

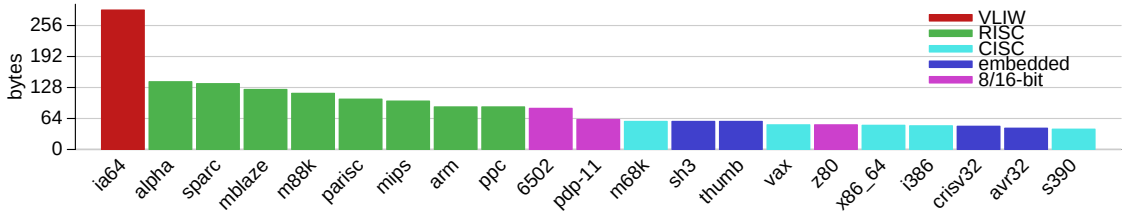

Figure A.5: Size of string searching code (unaligned load instructions help, since four bytes at arbitrary offsets can be compared at once. CISC architectures as well as avr32 and MIPS benefit)

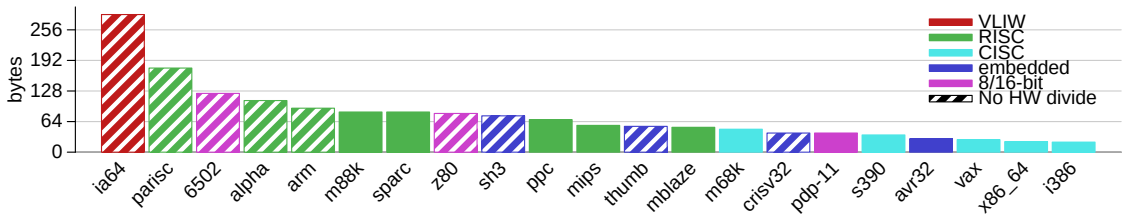

Figure A.6: Size of integer printing code (hardware divide helps code density)

Minimum instruction length: Short instruction encodings help most with respect to reducing density. Architectures with variable-length instructions, especially those with useful single-byte instructions (like x86 and VAX), can accomplish much work with little code. Fixed-length ISAs can be dense if all instructions are 16-bit (like AVR32 and SH3); RISCs with fixed 32-bit instructions generate less dense code; and the VLIW generates the least dense code of all platforms studied. Figure A.3's LZSS decompression clearly demonstrates this.

Number of integer registers: Having fewer registers reduces the number of bits needed to encode instructions, increasing code density. There is a tradeoff, in that having fewer registers generates more loads/stores from spilling in load-store architectures.

Virtual address of first instruction: Operating system design decisions affect code density. If the virtual address space is configured so programs start near the bottom of virtual memory, then a 16-bit constant is enough to point to a small program's entire memory. Constant 32-bit pointer loads are at least double the size of 16-bit loads on most architectures, and 64-bit pointer loads are even more wasteful. Using small system call numbers can help, too; avoiding large immediate constants saves space in executables.

Existence of a zero register: Zero registers are normally found in RISC architectures, so they tend to correlate with less dense code. A zero register can be simulated using one load instruction and sacrificing a register, so the feature offers few benefits with regards to code density.

Bit width: Having a narrower bit-width leads to denser code, mainly due to shorter immediate values for pointer loads and branch offsets.

Number of operations in instruction: Operation count directly affects the size of instruction encoding.

Year of introduction: Somewhat surprisingly, age does not correlate highly with code density. This is due to the many embedded architectures introduced recently.

Branch delay slots: Branch delay slots can decrease code density due to added nops. For our benchmark, slots can often be filled, so branch delay slots cause no problem.

Endianess: Endianess has little impact on code density unless the program operates on data in a non-native format.

Status flags: Upon completion of ALU operations, these flags (or condition codes) are set as side effects to indicate that the result was zero, negative, an overflow, etc. These flags can lead to denser code by eliminating the need for comparison instructions before conditional branches. Most RISC designs avoid status flags, as they add complexity and ordering dependencies to out-of-order processors.

Auto-increment addressing: Auto-increment addressing modes allow accessing consecutive memory addresses without requiring separate increment instructions. This is especially useful for accessing arrays, of which C strings are a subset. String copying and concatenation, as in Figure A.4, benefit from these instructions.

Unaligned memory access: Allowing unaligned loads and stores leads to smaller code, especially for string manipulation. Unaligned 16 and 32 bit loads permit arbitrary simultaneous access to consecutive bytes in memory. If alignment is enforced, achieving the same results requires a series of memory, shift, and logical operations. Results in Figure A.5 demonstrate benefits of this feature.

Hardware division: A hardware divide instruction is often slower than using the equivalent multiply by the reciprocal [59] or lookup table-based division routines, but it almost always takes fewer bytes in the instruction stream. Some architectures only implement single-bit division routines that require software pipelining; this can lead to less space-efficient code than otherwise undesirable algorithms such as iterative subtraction. Integer printing code benefits greatly from hardware divide, as in Figure A.6.

A.5 Density of Compiler-Generated Binaries

Hand-optimizing large programs in assembly language is impractical under most circumstances. We therefore evaluate compact code generation using more traditional methods. We choose to experiment with the x86 architecture due to its popularity and high code density.

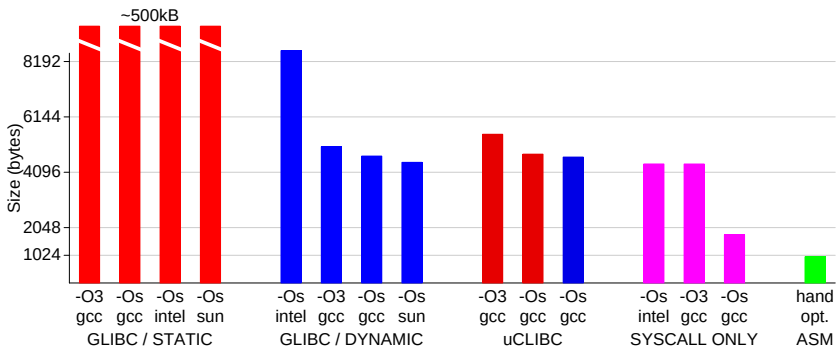

Figure A.7: Total size of generated executables, stripped of debugging information.

We use a variety of C compilers and libraries to determine how small an executable we can generate using off-the-shelf tools. We use the GNU gcc 4.2 compiler (gcc 4.1 for uClibc runs), the Intel C compiler version 9.1.038, and the SunStudio 12 compiler, all under Linux. We use GNU libc 2.7 and the embedded uClibc 0.9.27.

We experiment with different compiler optimizations. In general, we use `-O3`; this usually optimizes for maximum performance. We also evaluate `-Os`, which optimizes for size. In practice, resulting executables are very similar. The primary differences are lack of loop unrolling, use of the hardware divide instruction instead of the faster multiply by reciprocal method, lack of function inlining, and less aggressive padding of function entry points.

Figure A.7 shows that executable sizes vary by many orders of magnitude. This is because statically linked programs contain the entire C library, which represents an overhead of at least 450KB (when using glibc).

By writing code that avoids the C library (and using system calls directly), we obtain executables only twice as large as hand-optimized codes. The remain-

ing reasons for larger code are:

- setting up the stack frame pointer at function entry — this can be turned off with the compiler option `---fomit-frame-pointer`;
- writing back to memory using 32-bit constants — due to pointer aliasing issues the compiler must frequently write values to memory using 5-byte instructions. The optimized assembler avoids aliasing and places more values in registers;
- loading of constants inefficiently — there are various slow (but smaller) ways to load small constants on x86; and
- avoiding string instructions — the compiler simply does not use the x86 specialized string instructions.

A.6 Related Work

Most code density research addresses the compressibility of instruction code [158, 82, 39, 20, 80, 166, 85, 149, 160, 130, 27]. Usually what is compressed is compiler-generated RISC or VLIW code, with compression ratios typically in the 50-70% range. We show here that embedded and CISC ISAs yield smaller binaries than RISC. Adding compression to a RISC architecture likely negates the speed benefits and decoder simplicity that initially motivated the move away from CISC.

Previous work compares multiple architectures, but our work is unique in the number (21) considered. Kozuch and Wolfe [79] measure entropy and compressibility of six different architectures (VAX, MIPS, SPARC, m68k, RS6000 and

PowerPC). Hasegawa et al. [63] compare SH3 code density to that of code generated by gcc on 10 other platforms (m68k, IA32, i960, Sparclite, SPARC, MIPS, AMD29k, m88k, Alpha, and RS6000). They find results roughly similar to ours, though they find the SH3 architecture generates smaller code than the x86 and m68k by a small margin. Flynn, Mitchell, and Mulder [54] compare code density of synthetic architectures that do not model actual systems.

Phelan [123] investigates features added to Thumb-2 to increase code density. Thumb-2 uses specialized instructions for enhanced constant support, limited predication, and compare-against-zero. These are similar features to those we find useful for density in Section A.4. Halambi et al. [61] investigate the benefits of using a *reduced Instruction Set Architecture* (rISA), such as THUMB and MIPS-16. They test hypothetical architectures, finding that a hybrid approach unlike any current reduced architecture should perform best.

Massalin's Superoptimizer [93] cleverly generates extremely dense (and non-intuitive) m68k and IA32 code by exhaustive search, but it only operates on small blocks of code (i.e., it's a highly tuned peephole optimizer).

A.7 Conclusions and Future Work

A 1987 article by Chow and Horowitz [34] quotes an early MIPS-X design document:

> "The goal of any instruction format should be: 1. simple decode, 2. simple decode, and 3. simple decode. Any attempts at improved code density at the expense of CPU performance should be ridiculed

at every opportunity."

Two decades later, the debate between prioritizing code density versus decoder simplicity in ISAs continues.

We investigate code density of 21 different architectures, and find that very high density levels can be achieved with proper planning of an ISA. To thoroughly exploit ISA density there must be cooperation between the operating system, system libraries, and compiler. On the x86 architecture, even after eliminating the C library and choosing maximum compiler options, a factor of two in code density can still be realized by hand-optimizing the assembly code. This is much greater than the 25% average size difference between RISC and CISC codes.

New ISAs, especially embedded ones, are continually being developed. Now that FPGAs are powerful enough to contain competitive CPUs, this trend of creating custom ISAs will likely increase. To aid in this development, we show which architectural features contribute most to code density, but also show that the entire system stack must be optimized to avoid wasting an ISA's inherent potential for density.

Ongoing work applies some of what we have learned to much bigger benchmarks to see what the performance and power implications are of using smaller libraries and different compiler options on larger applications. We hope to raise awareness of the importance of code density on all modern architectures, not just those targeted for the embedded space.

APPENDIX B

CACHE LATENCIES

Tables B.1 and B.2 show cache latencies for various machines available in CSL. This is useful when judging how realistic cache settings are in simulators. The results were generated with the lmbench tool (http://www.bitmover.com/lmbench/), and were spot-checked against actual documentation to make sure the results were sane. Many of the older chips with extremely high L2 latencies have off-chip L2s.

Table B.1: L1 Cache latencies on Fusion group machines

Machine	CPU	Freq (MHz)	DL1 Latency (ns)	DL1 Latency (cycles)	DL1 Size	DL1 Assoc
sampaka12	P4 Xeon	2790	0.703	2	8k	4-way
cluizel	P4 Xeon	2787	0.724	2	8k	4-way
dolfin	P4	2394	0.853	2	8k	4-way
cluster	Core2 E5440	2820	1.066	3	32k	8-way
cluster-026	Core2 E5430	2660	1.128	3	32k	8-way
domori25	Pentium D	3463	1.155	4	16k	8-way
ithaca	P4 Xeon	2988	1.343	4	16k	8-way
tasse	Core2 Q6600	2399	1.250	3	32k	8-way
venchi	Phenom 9500	2212	1.360	3	64k	2-way
ps3	Cell	1591	1.545	2.5?	32k	?-way
old-milka	Athlon MP	1729	1.735	3	64k	2-way
tobler	Athlon XP	1663	1.804	3	64k	2-way
chocovic	Core T2300	1600	1.812	3	32k	8-way
atom-power	Atom N270	1597	1.899	3	24k	6-way
lindt	ARM v5te	1197	2.599	3	16k	4-way
valor	Niagara1	1000	3.115	3	8k	?-way
elrey09	Power3	375	5.335	2	64k	128-way
spruengli	Pentium III	545	5.497	3	16k	4-way
carnivore	USPARC II	359	5.597	2	16k	1-way
bmul	Alpha EV6	496	6.042	3	64k	2-way
hershey	MIPS R12k	300	6.630	2	32k	2-way
nestle	Pentium II	400	7.495	3	16k	4-way
perugina	MIPS R5k	178	11.500	2	32k	2-way
ancient	Pentium Pro	198	15.100	3	8k	2-way

Table B.2: L2 Cache latencies on Fusion group machines

Machine	CPU	Freq (MHz)	L2 Latency (ns)	(cycles)	L2 size
cluster	Core2 E5440	2820	5.365	15	6144k
cluster-026	Core2 E5430	2660	5.665	15	6144k
tasse	Core2 Q6600	2399	5.863	14	4096k
sampaka12	P4 Xeon	2790	6.568	18	512k
cluizel	P4 Xeon	2787	6.575	18	512k
venchi	Phenom 9500	2212	6.950	15	512k
dolfin	P4	2394	7.720	18	512k
domori25	Pentium D	3463	7.982	28	2048k
chocovic	Core T2300	1600	8.436	14	2048k
ithaca	P4 Xeon	2988	9.431	28	1024k
atom-power	Atom N270	1597	10.200	16	512k
old-milka	Athlon MP	1729	11.600	20	256k
tobler	Athlon XP	1663	12.000	20	256k
ps3	Cell	1591	12.600	20	512k
lindt	ARM v5te	1197	21.000	25	256k
valor	Niagara1	1000	22.100	22	3072k
carnivore	USPARC II	359	27.900	10	4096k
bmul	Alpha EV6	496	30.300	15	4096k
elrey09	Power3	375	32.000	12	512k
spruengli	Pentium III	545	33.000	18	512k
ancient	Pentium Pro	198	35.400	7	512k
hershey	MIPS R12k	300	47.300	14	2048k
nestle	Pentium II	400	55.000	22	512k
perugina	MIPS R5k	178	256.700	46	512k

APPENDIX C

INSTRUCTION COUNTS

This appendix contains retired instruction counts for various architectures, comparing simulators and hardware performance counters (if available). Retired instructions are a useful metric, as the results should be the same (within reason, see Chapter 4) across all implementations, including simulators. These tables show that our results are reasonable across independent implementations, including actual hardware and the various simulators.

Table C.1 shows Alpha retired instruction counts for SPEC CPU2000 on m5 and on Qemu.

Table C.2 shows MIPS retired instruction counts for an actual R12000 processor as well as Qemu.

Table C.3 shows PPC retired instruction counts for Qemu and Valgrind. Some preliminary performance counter results are available for a G3 processor, however the retired instruction counter on that architecture does not count many branch instructions (so call "folded" branch instructions) so the hardware undercounts by a large amount.

Table C.4 shows SPARC retired instruction counts on a Niagara processor as compared to Qemu for SPEC CPU2000. Tables C.5 and C.6 show SPARC retired instruction counts on a Niagara processor as compared to Qemu for SPEC CPU2006.

Table C.7 shows x86 retired instruction counts for Pin, Valgrind and Qemu as well as native Pentium D for SPEC CPU2000. Tables C.8 and C.9 show the

DBI results for SPEC CPU2006.

Table C.10 shows x86_64 retired instruction counts for Valgrind as well as m5 and native Pentium D for SPEC CPU2000.

Table C.1: Retired instructions for Alpha SPEC CPU2000, showing Qemu and m5 results.

Benchmark	Qemu	m5
perlbmk.mkrnd	1,852,185,707	1,852,370,276
gcc.expr	10,526,941,197	10,526,937,311
gcc.integrate	11,131,148,640	11,131,144,748
perlbmk.prfct	28,651,272,612	28,646,299,027
gcc.166	38,549,908,675	38,549,904,548
perlbmk.diff	43,466,596,591	43,465,983,649
gzip.log	48,291,591,874	48,291,602,124
gcc.scilab	54,995,351,679	54,995,349,608
mcf	58,172,989,262	58,173,013,914
art.110	64,796,575,048	64,796,606,371
eon.rushmeier	66,721,596,572	66,721,584,534
art.470	71,264,085,737	71,264,936,399
perlbmk.535	71,270,714,899	71,270,497,774
perlbmk.704	75,243,372,695	75,243,363,852
vpr.route	91,648,810,187	91,649,020,376
eon.cook	93,166,307,448	93,166,297,706
gzip.random	96,314,185,218	96,314,195,503
gzip.source	96,622,781,614	96,622,791,539
gcc.200	98,082,642,278	98,083,090,701
eon.kajiya	118,887,100,043	118,887,092,853
bzip2.source	119,407,524,410	119,407,527,861
vpr.place	122,094,665,532	n/a
gzip.graphic	122,633,026,085	122,633,036,370
perlbmk.957	124,615,156,000	124,614,368,257
perlbmk.850	144,811,182,497	144,811,615,781
bzip2.program	157,826,408,892	157,826,413,107
gzip.program	183,690,076,747	n/a
equake	184,570,930,271	184,571,385,450
bzip2.graphic	188,304,395,299	188,304,399,124
crafty	202,097,841,686	202,098,026,258
lucas	250,361,411,526	250,361,400,189
gap	282,645,795,759	282,645,790,314
swim	314,735,637,962	314,735,601,038
mesa	356,141,326,372	356,140,991,620
wupwise	441,495,751,933	441,495,740,707
galgel	445,002,076,017	445,002,059,655
facerec	455,849,231,313	n/a
parser	457,300,531,283	457,326,609,655
twolf	471,369,350,904	471,369,355,199
fma3d	489,141,407,181	489,141,392,542
apsi	530,748,539,989	530,748,529,384
ammp	589,337,621,794	589,337,650,371
applu	593,502,981,717	593,502,909,954
sixtrack	660,176,134,816	660,175,855,578
mgrid	674,760,931,332	674,760,591,777
vortex.1	n/a	n/a
vortex.2	n/a	n/a
vortex.3	n/a	n/a

Table C.2: Retired instructions for MIPS SPEC CPU2000, showing both Qemu and actual hardware.

Benchmark	MIPS R12000	Qemu
perlbmk.makernd	2,119,184,698	2,114,256,173
gcc.expr	10,231,747,780	10,379,335,965
gcc.integrate	10,877,917,321	11,029,085,971
perlbmk.perf	34,397,475,736	34,291,318,887
gcc.166	38,345,161,913	38,937,127,796
gzip.log	39,177,926,195	39,176,934,404
perlbmk.diffml	49,045,516,380	48,866,727,288
gcc.scilab	53,424,569,071	54,082,349,765
art.110	59,896,479,904	59,825,327,249
art.470	65,822,125,896	65,744,913,201
mcf	76,703,646,467	74,897,119,306
gzip.random	78,014,071,182	78,006,505,814
eon.rushmeier	80,494,543,229	80,416,672,726
perlbmk.535	80,842,440,768	80,746,160,901
gzip.source	83,418,955,967	83,417,498,631
perlbmk.704	84,992,707,539	84,872,174,178
gcc.200	92,366,416,553	93,545,622,835
vpr.route	97,113,968,526	95,326,686,529
gzip.graphic	103,561,100,092	103,554,490,672
bzip2.source	112,923,929,153	112,719,582,700
eon.cook	113,042,582,300	112,907,012,118
bzip2.program	127,535,065,353	127,404,501,780
vpr.place	132,437,075,241	131,368,316,663
perlbmk.957	140,872,244,535	140,683,409,440
eon.kajiya	143,533,032,882	143,431,929,088
bzip2.graphic	147,248,822,502	146,863,157,720
vortex.1	148,901,262,048	148,334,470,156
perlbmk.850	164,789,137,006	164,615,395,390
vortex.3	166,032,274,034	165,390,346,140
gzip.program	166,615,637,739	166,611,066,062
vortex.2	170,449,198,654	169,923,949,114
equake	230,432,212,869	230,390,078,651
gap	276,977,290,268	273,263,441,277
crafty	280,337,735,418	278,860,598,584
swim	333,553,099,917	329,147,033,985
twolf	361,806,897,205	356,817,153,112
mesa	366,180,560,698	360,389,956,020
lucas	375,444,997,702	372,854,255,435
facerec	423,358,395,046	416,148,523,943
wupwise	447,314,166,380	447,259,587,502
galgel	491,136,359,115	487,290,327,879
parser	490,008,576,749	487,974,794,897
apsi	568,246,899,533	564,420,021,462
fma3d	616,533,917,707	610,504,190,176
mgrid	701,712,476,632	701,661,166,808
applu	728,501,949,684	728,037,101,288
ammp	729,508,483,198	728,520,511,603
sixtrack	missing	missing

Table C.3: Retired instructions for PPC SPEC CPU 2000, showing Qemu and Valgrind results.

Benchmark	Qemu	Valgrind
gcc.expr	9,190,508,966	9,190,408,368
gcc.integrate	9,411,432,531	9,411,456,014
gcc.166	31,507,819,529	31,507,331,371
gzip.log	31,513,650,235	31,513,659,481
gcc.scilab	47,597,841,628	47,596,075,897
art.110	50,150,252,743	50,150,283,130
mcf	50,709,526,816	50,709,625,911
art.470	54,693,010,934	54,693,752,821
eon.rushmeier	54,820,758,341	54,820,766,094
gzip.source	62,856,871,039	62,856,880,266
gzip.random	63,122,070,324	63,122,079,446
eon.cook	76,481,665,935	76,481,673,844
gzip.graphic	80,803,892,856	80,803,902,671
gcc.200	82,004,825,387	81,998,587,706
vpr.route	87,742,570,724	87,743,079,655
bzip2.source	95,017,636,258	95,017,641,106
eon.kajiya	98,286,987,225	98,286,994,837
bzip2.program	115,072,240,303	115,072,245,682
gzip.program	120,028,837,910	120,028,848,631
vpr.place	121,600,539,123	121,600,943,120
equake	131,373,993,006	131,374,564,801
bzip2.graphic	136,166,466,788	136,166,471,812
lucas	178,271,603,649	178,271,598,361
swim	219,242,905,224	219,242,591,398
gap	235,654,361,137	235,654,361,261
crafty	n/a	245,395,327,943
mesa	268,532,620,295	268,532,873,638
facerec	273,239,656,811	273,239,663,554
galgel	296,856,753,462	296,856,724,267
wupwise	310,764,127,217	310,764,121,790
twolf	312,615,581,513	312,615,639,218
perlbmk.mkrnd	337,914,199,076	337,955,381,950
apsi	356,063,828,623	356,064,354,647
applu	385,545,778,293	385,545,377,856
fma3d	392,497,217,043	392,497,240,711
parser	393,180,431,370	393,076,146,215
ammp	408,057,478,840	408,058,156,433
mgrid	422,477,381,101	422,475,227,875
sixtrack	n/a	958,356,774,666

Table C.4: Retired instructions for SPARC SPEC CPU2000, showing actual hardware and Qemu results.

Benchmark	niagara	Qemu	% diff
perlbmk.mkrnd	1,404,947,767	1,404,994,700	0.0033%
gcc.expr	7,438,524,056	7,439,822,058	0.0174%
gcc.integrate	7,496,590,557	7,496,932,953	0.0046%
perlbmk.perf	22,090,018,022	22,117,246,402	0.1233%
gcc.166	24,096,748,165	24,099,211,420	0.0102%
gzip.log	32,220,222,634	32,220,258,877	0.0001%
perlbmk.diffml	32,527,608,598	32,552,061,978	0.0752%
gcc.scilab	38,943,582,033	38,954,242,610	0.0274%
perlbmk.535	52,797,417,192	52,812,588,889	0.0287%
art.110	55,919,697,739	55,919,809,366	0.0002%
perlbmk.704	56,138,421,439	56,155,913,315	0.0312%
eon.rushmeier	59,734,334,097	59,754,206,796	0.0333%
art.470	61,320,265,673	61,320,378,433	0.0002%
gzip.source	64,028,560,356	64,028,627,716	0.0001%
mcf	66,952,070,888	66,955,397,677	0.0050%
gcc.200	69,304,422,957	69,323,496,419	0.0275%
gzip.random	72,129,328,188	72,129,329,697	0.0000%
vpr.route	80,003,670,096	80,004,767,633	0.0014%
eon.cook	81,546,300,471	81,582,283,446	0.0441%
gzip.graphic	84,794,373,750	84,794,614,880	0.0003%
bzip2.source	85,705,886,914	85,705,888,072	0.0000%
perlbmk.957	93,025,850,757	93,047,199,542	0.0229%
bzip2.program	104,202,484,846	104,202,485,985	0.0000%
vortex.1	104,232,857,833	104,324,773,140	0.0882%
eon.kajiya	105,264,967,103	105,318,413,139	0.0508%
perlbmk.850	107,063,868,164	107,084,662,958	0.0194%
vortex.2	112,156,726,713	112,244,643,448	0.0784%
vpr.place	115,176,477,366	115,176,656,963	0.0002%
vortex.3	116,025,640,113	116,128,639,345	0.0888%
bzip2.graphic	120,800,611,046	120,800,612,184	0.0000%
gzip.program	123,680,369,194	123,680,901,121	0.0004%
equake	151,889,001,382	151,891,686,411	0.0018%
crafty	206,305,279,572	206,315,053,194	0.0047%
gap	210,517,827,839	210,558,299,855	0.0192%
swim	240,514,963,796	240,515,886,318	0.0004%
lucas	266,811,475,823	266,811,478,992	0.0000%
twolf	305,337,539,213	305,339,039,589	0.0005%
facerec	326,679,452,956	326,696,166,196	0.0051%
fma3d	341,030,439,528	341,075,610,275	0.0132%
wupwise	341,661,289,809	341,661,294,714	0.0000%
mesa	346,548,676,244	346,551,169,132	0.0007%
ammp	349,598,884,109	349,606,001,911	0.0020%
parser	355,866,671,084	356,177,932,828	0.0875%
galgel	384,843,413,509	384,843,478,318	0.0000%
apsi	404,410,763,702	404,412,372,555	0.0004%
applu	483,553,093,892	483,553,210,988	0.0000%
mgrid	523,649,830,576	523,650,635,035	0.0002%
sixtrack	531,777,045,890	531,779,537,683	0.0005%

Table C.5: Retired instructions for SPARC SPEC CPU2006, showing actual hardware and Qemu results (part 1)

Benchmark	niagara	Qemu
gcc.scilab	59,448,490,062	59,478,674,491
gcc.166	80,966,297,067	81,044,697,421
gcc.cp-decl	106,554,251,986	106,581,071,958
gcc.expr	118,262,219,122	118,493,954,045
gcc.c-typeck	138,438,742,077	138,469,255,376
gcc.200	154,653,416,830	154,717,033,840
perlbench.spam	155,506,014,368	155,625,157,413
gcc.expr2	160,804,486,540	161,150,848,871
gcc.s04	172,935,550,308	172,992,456,986
gcc.g23	196,380,751,261	196,400,952,072
bzip2.chicken	202,417,012,643	202,417,033,026
gobmk.trevorc	260,110,020,234	260,217,689,528
gobmk.13x13	260,461,398,599	260,568,405,602
bzip2.liberty	327,264,767,657	327,264,785,337
gobmk.score2	371,492,860,260	371,607,697,991
gobmk.trevord	373,599,357,159	373,751,619,245
h264ref.f_main	383,498,922,741	383,529,239,000
perlbench.diff	394,828,296,723	395,404,360,292
bzip2.combined	414,138,036,982	414,144,817,141
soplex.pds-50	418,216,583,066	418,140,134,600
mcf	439,161,059,237	439,179,344,637

Table C.6: Retired instructions for SPARC SPEC CPU2006, showing actual hardware and Qemu results (part 2)

Benchmark	niagara	Qemu
bzip2.source	525,214,607,479	525,214,762,807
h264ref.f_base	574,295,012,515	574,297,914,365
omnetpp	636,530,417,491	639,102,782,063
bzip2.program	659,510,973,841	659,511,129,188
gobmk.nngs	690,739,847,792	691,017,039,711
perlbench.split	760,225,673,600	760,357,819,246
bzip2.html	789,783,660,823	789,783,485,905
gamess.h2ocu2	971,364,312,622	971,365,469,106
povray	1,025,406,672,709	1,025,860,891,617
milc	1,162,528,869,104	1,162,528,877,675
hmmer.nph3	1,171,717,651,459	1,171,713,503,032
gromacs	1,734,832,898,895	1,734,834,096,917
xalancbmk	1,207,787,642,280	1,210,553,356,879
gamess.cytosine	1,307,955,048,365	1,307,955,638,267
lbm	1,682,804,008,345	1,682,806,635,812
hmmer.retro	2,478,793,495,266	2,478,795,502,553
sjeng	2,953,423,604,499	2,953,623,143,659
GemsFDTD	3,129,653,180,977	3,129,659,014,007
libquantum	3,347,218,538,866	3,347,218,540,333
h264ref.s_main	3,427,244,367,161	3,427,523,271,405
sphinx3	3,527,964,118,035	3,528,003,700,847
cactusADM	3,719,473,172,837	n/a
tonto	3,769,023,136,632	3,772,119,969,618
namd	3,824,795,643,390	3,824,796,483,411
gamess.tri	4,092,000,060,424	4,092,002,135,106
bwaves	4,249,238,621,349	4,249,236,595,963
leslie3d	5,712,418,884,905	5,712,419,557,221
calculix	7,587,333,128,722	n/a
dealII	2,081,474,987,248	n/a
soplex.ref	452,326,930,326	n/a
wrf	n/a	n/a
zeusmp	2,395,312,908,679	n/a
astar.BigLakes	n/a	n/a
astar.rivers	n/a	n/a

Table C.7: Retired instructions for x86 SPEC CPU2000, showing both Qemu and actual hardware.

Benchmark	Pentium D	Pin	Qemu	Valgrind
188.ammp	333,169,333,372	333,169,294,670	333,169,294,696	333,169,329,139
173.applu	554,510,033,405	554,509,978,381	554,509,978,455	554,509,978,924
301.apsi	648,607,278,050	648,607,219,730	648,607,218,356	648,607,225,337
179.art 110	117,967,839,911	117,967,839,150	117,967,839,198	58,632,609,034
179.art 470	121,326,001,662	121,326,000,207	121,326,000,255	64,317,418,082
256.bzip2 graphic	117,528,983,508	117,528,935,447	117,528,935,461	117,528,935,918
256.bzip2 program	103,252,292,827	103,252,244,840	103,252,244,854	103,252,245,311
256.bzip2 source	86,640,101,418	86,640,053,044	86,640,053,064	86,640,053,515
186.crafty	215,657,884,011	215,657,814,911	215,657,814,944	215,657,815,392
252.eon cook	85,146,651,778	85,146,645,565	85,146,645,795	85,146,647,511
252.eon kajiya	109,342,530,157	109,342,523,528	109,342,523,841	109,342,278,681
252.eon rushmeier	62,973,702,423	62,973,695,532	62,973,695,924	62,973,696,148
183.equake	144,985,852,691	144,985,830,752	144,985,830,784	144,985,809,902
187.facerec	309,900,303,948	309,897,997,695	309,897,997,844	309,897,997,983
191.fma3d	320,946,898,097	320,946,792,622	320,946,792,780	320,946,772,563
178.galgel	370,730,832,412	370,730,602,525	370,730,602,672	370,916,153,512
254.gap	221,616,787,940	221,616,650,209	221,616,650,177	221,616,650,640
176.gcc 166	22,310,940,703	22,310,842,538	22,310,842,738	22,311,064,104
176.gcc 200	72,618,452,686	72,618,200,323	72,618,200,652	72,618,892,100
176.gcc expr	7,287,040,224	7,286,992,242	7,286,992,508	7,287,101,582
176.gcc integrate	7,295,131,438	7,295,099,630	7,295,099,672	7,295,204,413
176.gcc scilab	39,177,416,341	39,177,182,848	39,177,182,732	39,176,446,405
164.gzip graphic	73,929,735,977	73,929,689,944	73,929,689,958	73,929,690,409
164.gzip log	29,339,108,465	29,339,062,611	29,339,062,625	29,339,063,076
164.gzip program	105,592,070,042	105,592,024,103	105,592,024,117	105,592,024,568
164.gzip random	60,368,090,965	60,368,044,987	60,368,044,996	60,368,045,458
164.gzip source	56,026,942,353	56,026,896,002	56,026,896,011	56,026,896,473
189.lucas	299,119,908,646	299,119,860,603	299,119,860,756	299,141,824,974
181.mcf	69,384,440,147	69,385,080,146	69,385,080,176	69,385,080,617
177.mesa	282,923,973,831	282,923,962,440	282,923,962,466	282,923,962,929
172.mgrid	502,690,354,308	502,690,322,067	502,690,322,141	502,690,797,081
197.parser	372,095,799,952	372,120,124,402	372,139,619,752	372,100,648,992
253.perlbmk 535	54,500,531,177	54,501,566,628	54,500,611,900	54,494,161,136
253.perlbmk 704	57,746,134,915	57,747,635,083	57,747,177,503	57,740,259,411
253.perlbmk 957	95,767,811,078	95,767,410,497	95,768,071,803	95,757,389,828
253.perlbmk 850	110,760,424,439	110,760,912,962	110,760,662,259	110,748,559,089
253.perlbmk diffmail	32,815,507,767	32,809,491,692	32,811,408,056	32,805,199,186
253.perlbmk mkrnd	1,265,082,975	1,265,125,871	1,265,119,126	1,265,122,789
253.perlbmk perfect	21,360,587,029	21,358,546,517	21,359,077,277	21,359,269,806
200.sixtrack	907,227,353,765	907,226,845,666	907,226,845,733	907,226,834,827
171.swim	301,163,912,858	301,163,859,925	301,163,859,993	301,163,890,220
300.twolf	311,868,478,360	311,868,472,123	311,868,472,155	311,868,472,603
255.vortex 1	144,373,942,830	144,373,882,650	144,373,882,668	144,369,991,734
255.vortex 2	162,519,416,945	162,519,362,767	162,519,362,785	162,517,282,072
255.vortex 3	160,888,128,313	160,888,065,924	160,888,065,942	160,890,052,476
175.vpr place	110,294,409,743	110,294,407,436	110,294,407,461	110,294,407,909
175.vpr route	93,441,428,087	93,441,408,681	93,441,408,697	93,441,437,858
168.wupwise	502,204,546,200	502,204,501,014	502,204,501,084	502,204,501,530

Table C.8: Retired instructions for x86 SPEC CPU2006, showing Pin, Valgrind, and Qemu and Pentium D (part 1).

Benchmark	Pentium D	Pin	Qemu	Valgrind
473.astar BigLakes	435,510,885,945	435,510,704,863	435,510,704,889	435,571,189,405
473.astar rivers	870,943,327,262	870,943,274,640	870,943,274,666	870,945,429,505
410.bwaves	2,495,857,175,894	2,495,855,313,228	2,488,693,693,543	2,497,901,489,742
401.bzip2 chicken	199,232,656,452	199,232,627,434	199,232,627,466	199,232,627,914
401.bzip2 combined	364,136,091,950	364,135,929,341	364,135,929,355	364,135,929,812
401.bzip2 html	706,417,018,345	706,416,797,441	706,416,797,473	706,416,797,921
401.bzip2 liberty	346,361,794,588	346,361,765,172	346,361,765,204	346,361,765,652
401.bzip2 program	593,333,086,611	593,332,865,463	593,332,865,483	593,332,865,934
401.bzip2 source	452,012,609,560	452,012,385,798	452,012,385,812	452,012,386,269
436.cactusADM	3,149,915,322,930	3,149,914,624,261	3,149,914,624,429	3,149,914,680,629
454.calculix	8,687,262,977,320	8,687,259,261,309	8,687,259,304,233	8,687,445,799,664
447.dealII	2,334,573,872,592	2,334,572,223,794	2,334,572,223,809	2,330,760,559,334
416.gamess cytosine	1,143,014,974,777	1,143,014,915,456	1,143,014,915,621	1,142,857,629,518
416.gamess h2ocu2	867,682,898,659	867,682,786,674	867,682,786,822	867,681,851,546
416.gamess triazolium	4,215,197,021,876	4,215,196,654,946	4,215,196,655,094	4,215,183,173,721
403.gcc 166	85,720,786,510	85,719,045,618	85,719,045,707	85,729,929,027
403.gcc 200	166,630,914,986	166,629,909,806	166,629,909,895	166,629,517,693
403.gcc c-typeck	140,819,919,763	140,813,681,279	140,813,681,371	140,836,875,542
403.gcc cp-decl	109,542,581,703	109,541,663,142	109,541,663,224	109,553,354,039
403.gcc expr	118,136,049,520	118,131,076,133	118,131,076,225	118,152,895,761
403.gcc expr2	160,294,450,226	160,288,195,981	160,288,196,080	160,319,263,901
403.gcc g23	193,775,955,105	193,769,398,187	193,769,398,269	193,795,174,499
403.gcc s04	179,205,087,128	179,202,306,373	179,202,306,455	179,225,687,180
403.gcc scilab	64,696,677,236	64,696,579,050	64,696,579,111	64,697,188,635

Table C.9: Retired instructions for x86 SPEC CPU2006, showing Pin, Valgrind, and Qemu and Pentium D (part 2).

Benchmark	Pentium D	Pin	Qemu	Valgrind
445.gobmk 13x13	238,220,175,611	238,220,161,269	238,220,161,299	238,220,161,795
445.gobmk nngs	631,487,346,762	631,487,324,668	631,487,324,687	631,487,325,199
445.gobmk score2	345,153,272,758	345,153,264,793	345,153,264,841	345,153,265,332
445.gobmk trevorc	236,505,606,986	236,505,585,769	236,505,585,812	236,505,586,316
445.gobmk trevord	340,188,792,640	340,188,777,068	340,188,777,111	340,188,777,615
459.GemsFDTD	2,511,548,122,944	2,511,544,724,520	2,511,544,724,652	2,511,544,820,559
435.gromacs	2,929,271,991,015	2,929,271,975,841	2,929,271,975,828	2,929,271,976,443
464.h264ref forebase	564,679,796,012	564,679,752,397	564,679,752,405	564,679,821,693
464.h264ref foremain	323,101,393,242	323,101,365,998	323,101,366,009	323,101,161,152
464.h264ref sss	2,814,673,346,854	2,814,673,203,792	2,814,673,203,803	2,814,672,542,474
456.hmmer nph3	1,039,884,719,301	1,039,884,652,135	1,039,884,652,113	1,039,884,652,662
456.hmmer retro	2,212,798,693,606	2,212,798,691,472	2,212,798,691,507	2,212,798,691,985
470.lbm	1,495,737,680,772	1,495,737,572,643	1,495,737,572,670	1,495,736,273,152
437.leslie3d	2,534,171,419,277	2,534,170,033,329	2,534,170,033,474	2,534,170,033,260
462.libquantum	3,884,593,324,266	3,884,593,087,057	3,884,593,087,091	3,884,593,087,595
429.mcf	449,894,848,644	449,895,233,570	449,895,233,605	449,895,234,103
433.milc	1,386,822,701,292	1,386,801,295,324	1,386,801,294,849	1,386,800,309,412
444.namd	2,895,739,745,975	2,895,739,724,790	2,895,739,724,842	2,895,739,729,006
471.omnetpp	764,012,416,098	764,012,359,528	764,012,359,553	762,846,798,782
400.perlbench checkspam	148,065,633,450	148,067,449,377	148,061,319,677	148,020,669,811
400.perlbench diffmail	401,910,091,346	401,888,866,913	401,932,464,865	401,831,332,925
400.perlbench splitmail	714,326,937,255	714,290,352,654	714,309,327,813	714,461,865,506
453.povray	1,204,156,309,806	1,204,157,467,167	1,204,159,849,673	1,204,125,183,903
458.sjeng	2,530,950,089,415	2,530,950,014,161	2,530,950,014,181	2,530,950,014,688
450.soplex pds-50	450,970,473,747	450,957,516,054	450,959,827,905	476,893,662,178
450.soplex ref	459,067,223,899	459,034,016,857	459,020,702,694	477,057,983,713
482.sphinx3	2,827,878,655,277	2,827,878,538,864	2,827,878,538,664	2,828,020,092,268
465.tonto	2,895,603,788,430	2,895,583,404,681	2,895,626,773,725	2,895,531,451,323
481.wrf	4,117,176,411,328	4,117,161,034,700	4,117,161,034,831	4,116,831,956,200
483.xalancbmk	1,313,433,329,271	1,313,431,575,521	1,313,431,575,551	1,314,798,555,882
434.zeusmp	2,397,609,296,047	n/a	n/a	n/a

Table C.10: Retired instructions for x86_64 SPEC CPU2000, showing both Qemu and actual hardware.

Benchmark	Pentium D	m5	Valgrind
perlbmk.mkrnd	1,090,919,227	1,090,879,129	1,090,746,089
gcc.expr	7,350,887,801	7,257,774,662	7,258,023,131
gcc.integrate	7,698,302,743	7,598,617,570	7,597,927,209
perlbmk.pfct	19,654,889,034	19,649,264,066	19,674,125,598
gcc.166	26,258,578,150	26,053,572,133	26,053,249,578
gzip.log	27,720,223,414	27,629,578,439	27,630,555,769
art.110	37,684,130,154	37,684,106,361	37,684,089,804
gcc.scilab	39,085,872,433	38,718,233,041	38,719,744,344
art.470	41,815,575,277	41,815,549,206	41,814,960,116
eon.rushmeier	46,652,449,332	46,652,438,765	46,652,447,077
mcf	47,178,238,767	47,178,770,487	47,178,758,942
gzip.random	50,716,078,217	50,552,564,398	50,553,545,097
eon.cook	59,432,883,084	59,432,871,622	59,432,880,124
gzip.source	63,638,496,739	63,533,923,887	63,534,804,993
vpr.route	65,842,168,801	65,842,101,031	65,842,410,972
gzip.graphic	66,140,686,787	65,984,284,025	65,985,226,242
gcc.200	69,752,973,526	69,333,015,398	69,350,744,008
bzip2.source	75,737,059,115	75,736,212,867	75,737,065,461
eon.kajiya	79,548,196,338	79,548,182,772	79,548,189,789
vpr.place	91,801,882,750	91,627,577,007	91,801,833,351
equake	91,831,665,346	91,831,629,328	91,831,292,111
bzip2.program	92,195,189,138	92,194,260,068	92,195,239,731
bzip2.graphic	104,716,089,604	104,715,201,159	104,716,114,878
gzip.program	134,301,541,033	134,183,019,555	134,184,027,716
crafty	140,491,641,621	140,491,608,144	140,491,506,813
gap	183,443,821,679	183,443,733,395	183,443,755,451
lucas	205,651,052,365	205,650,963,195	205,650,990,335
swim	211,145,979,309	211,145,850,745	211,145,887,898
mesa	225,141,182,114	225,141,105,441	225,141,115,104
facerec	249,466,728,521	249,465,506,605	249,433,555,885
fma3d	252,621,825,649	252,621,687,157	252,621,712,799
parser	263,269,230,283	263,269,185,444	263,218,164,789
galgel	265,315,494,177	265,315,409,019	265,319,397,124
ammp	282,273,753,920	282,273,684,014	282,273,805,462
twolf	294,395,392,631	294,395,327,575	294,395,331,989
mgrid	317,902,282,935	317,901,442,889	317,901,782,490
applu	329,640,061,785	329,639,906,447	329,639,978,210
apsi	335,998,339,268	335,998,752,850	335,998,224,351
wupwise	360,553,449,666	360,553,370,094	360,553,381,385
sixtrack	542,751,559,882	542,751,311,580	542,751,677,787
perlbmk.535	n/a	n/a	n/a
perlbmk.704	n/a	n/a	n/a
perlbmk.850	n/a	n/a	n/a
perlbmk.957	n/a	n/a	n/a
perlbmk.diff	n/a	n/a	n/a
vortex.1	n/a	n/a	n/a
vortex.2	n/a	n/a	n/a
vortex.3	n/a	n/a	n/a

APPENDIX D

SIMULATION TIMINGS

This appendix contains details of how long various simulators take to simulate the SPEC CPU2000 benchmarks.

These results are only approximate. Times for simulators were measured on the domori cluster (3.46GHz Pentium D) with varying loads. These measurements were taken over a 5 year span, the operating systems were updated in that time, and the various simulators were also updated. Some points are missing, either due to broken simulators, cluster crashes, or broken toolchains. The `vortex` and some of the `perlbmk` benchmarks are missing, as they do not run out of the box with modern compilers on our comparison Pentium D machine.

Often multiple runs were taken, in that case the lowest value was chosen. For the native runs, at least three runs were made, with the middle chosen as the reference time. The average given is the weighted average across all benchmarks in the suite. If benchmarks are missing, the comparison is between that subset of benchmarks that completed.

Be careful if using these numbers for anything but rough estimates or orders-of-magnitude comparisons.

An overall summary can be found in Table D.1.

Table D.3 shows Alpha results. The binaries were compiled with gcc 4.2 using `-O2` optimization. The Qemu results are from git Qemu as of 11 January 2010 with a few extra patches applied to enable proper floating-point and fstat64 support. The many missing points for sim-alpha are due to that simulator not

supporting modern Linux binaries.

Table D.4 shows MIPS results. The binaries used are the pre-compiled ones from the SESC website. Care needs to be taken, as those binaries are incomplete (not all are available) and some of them, most notably `gzip`, have been modified for shorter runtime. This is why the `gzip` benchmarks seem to run faster on the older R12k machine than on the modern Pentium D machine. qemu_cache is Qemu custom-patched and feeding into the Dinero cache simulator.

Table D.5 shows SPARC results. qemu_bbv is Qemu patched to generate basic block vectors, hence a bit slower than stock Qemu. The niagara system shown for comparison has a simple FPU unit, which is why the floating point benchmarks perform relatively poorly.

Table D.6 shows x86 results for SPEC CPU2000. These are using pinkit pin-2.0-10520-gcc.4.0.0-ia32-linux, qemu 0.9.1 and Valgrind 3.3.0. According to the pin 2005 PLDI paper [87] the `perlbmk` and `gcc` slowdowns are because there is not much code reuse, so the jit overhead is higher. Integer codes perform worse due to the large number of indirect or unpredictable jumps. The m5 results are incomplete, as currently m5 x86 support is missing x87 floating point support. In many cases the 32-bit code performs better than the 64-bit code. Table D.2 breaks this out for the `sixtrack` benchmark. The difference seems to be inherent in the 64-bit aspect of the code, as running on different implementations shows the same problem, and turning on SSE instructions does not help (SSE instructions are limited to 64-bit floating point, so issues due to 80-bit floating point would be uncovered that way). The cause of this is still under investigation.

Table D.1: Summary of slowdown compared to Pentium D node running x86_64 binaries.

Arch.	Method	Minimum Slowdown		Maximum Slowdown		Weighted Average
Alpha	Alpha 21264	3.08	mcf	12.21	galgel	5.88
	qemu-alpha	3.67	mcf	261.04	mgrid	61.19
	m5-nocache	105.82	mcf	2696.46	mgrid	1312.59
	sim-alpha	617.19	mcf	8250.56	gap	3772.03
MIPS	MIPS R12k	0.65	gzip	22.95	swim	10.01
	qemu_cache	34.92	mcf	966.31	mgrid	376.21
	sesc	377.90	gzip	12435.67	mgrid	3909.04
SPARC	SPARC niagara	2.25	mcf	78.13	mgrid	20.95
	qemu_bbv	1.75	mcf	181.18	mgrid	44.58
x86	Pentium D	0.25	sixtrk	4.00	perl.rnd	0.97
	pin	1.47	sixtrk	47.00	perl.rnd	8.30
	qemu	2.69	mcf	84.38	wupwise	26.50
	valgrind	4.08	mcf	99.52	eon.cook	32.10
	m5-nocache	2943.13	gcc.166	7495.53	gap	5624.99
x86_64	pin baseline	0.89	sixtrk	4.50	gcc.expr	1.27
	qemu baseline	1.31	mcf	20.59	eon.cook	8.78
	valgrind baseline	1.64	mcf	42.44	mesa	6.92
	pin_cache	14.31	mcf	352.66	wupwise	101.68
	qemu_cache	50.44	mcf	17936.83	eon.cook	1036.74
	valgrind_cache	5.73	mcf	97.77	perl.pfct	29.13
	m5-nocache	358.71	mcf	6784.89	eon.rush	2694.39
	m5-cache	841.46	sixtrk	7104.07	wupwise	2882.13
	ptlsim	1070.03	mcf	14398.58	wupwise	7534.09

Table D.7 shows x86_64 results for simulations. The m5 version is the current mercurial tree as of 10 January 2010, with a few additional patches. Benchmarks are compiled with `-O3 -msse3 -funroll-all-loops -ffast-math` on gcc 4.3.2. Table D.8 shows unmodified x86_64 DBI results, showing the fastest possible speed. Valgrind is version 3.5, Qemu is the git tree as of 10 December 2009. Table D.9 shows x86_64 DBI results when including a cache simulator. The Qemu results are shown feeding the Dinero cache simulator via a named pipe with addresses truncated to 32-bits. The pin and Valgrind results are using the cache simulators that come with the tools, while simulating a Phenom-like cache infrastructure.

Table D.2: x86 32-bit versus 64-bit run time anomaly for `sixtrack`. Some benchmarks perform markedly worse when compiled as 64-bit.

	Pentium D	Phenom	Core2 Q6600
32-bit -O2	4:18	4:52	3:21
64-bit -O2	20:12	23:34	19:14
64-bit -msse3	23:00	37:08	18:57

Table D.3: Elapsed times for running the SPEC CPU 2000 benchmarks on various Alpha simulators. *domori* is time on our reference Pentium D machine. *bmul* is an actual Alpha 21264 system.

Benchmark	domori	bmul	qemu	m5-nocache	sim-alpha
perlbmk.makerand	1s	5s	21s	23m09s	n/a
gcc.integrate	5s	20s	1m45s	1h36m42s	6h12m38s
gcc.expr	4s	19s	1m51s	1h33m20s	6h04m18s
gzip.log	13s	1m27s	5m04s	6h16m20s	n/a
gcc.166	23s	1m30s	5m18s	5h13m42s	20h51m23s
eon.rushmeier	18s	1m38s	22m57s	9h32m53s	1d13h19m29s
gcc.scilab	23s	1m40s	11m02s	8h11m27s	1d08h16m48s
gzip.random	24s	2m35s	10m40s	13h34m50s	n/a
gzip.source	29s	2m50s	11m51s	12h56m51s	n/a
gzip.graphic	29s	2m56s	12m43s	18h03m23s	2d09h32m45s
gcc.200	40s	3m00s	17m09s	14h27m11s	2d21h34m53s
eon.kajiya	34s	3m07s	40m48s	17h34m35s	n/a
eon.cook	22s	3m09s	31m09s	13h07m53s	2d01h30m28s
bzip2.source	40s	3m38s	13m00s	17h07m12s	n/a
art.110	1m15s	4m24s	29m31s	8h08m28s	1d03h59m01s
bzip2.program	41s	4m24s	15m06s	22h34m37s	n/a
gzip.program	52s	4m43s	19m28s	23h53m45s	n/a
art.470	1m22s	4m57s	26m30s	8h51m15s	1d06h10m39s
vpr.place	1m28s	6m13s	31m21s	n/a	n/a
vpr.route	1m25s	6m20s	22m56s	15h47m11s	n/a
crafty	1m12s	6m21s	32m58s	1d12h58m47s	4d11h59m51s
bzip2.graphic	51s	8m00s	16m43s	1d02h36m04s	n/a
mesa	1m36s	8m44s	1h52m33s	2d03h57m47s	n/a
gap	1m13s	9m34s	38m51s	2d00h09m00s	6d23h18m11s
wupwise	1m40s	9m59s	3h12m48s	2d10h28m58s	7d09h32m35s
lucas	2m02s	10m07s	2h21m56s	1d09h02m13s	2d20h01m11s
equake	1m09s	10m13s	1h27m57s	1d02h20m24s	3d12h05m15s
sixtrack	2m46s	13m23s	8h18m57s	3d04h58m54s	n/a
facerec	3m56s	14m15s	2h14m20s	n/a	n/a
mcf	5m09s	15m52s	18m53s	9h04m58s	2d04h58m32s
applu	2m42s	15m57s	5h06m50s	3d04h53m25s	6d04h57m11s
fma3d	5m31s	17m57s	3h47m10s	2d21h39m19s	n/a
ammp	2m47s	18m35s	5h38m13s	3d10h47m49s	7d06h45m53s
mgrid	1m52s	19m42s	8h07m16s	3d11h53m24s	8d12h12m06s
galgel	1m54s	23m12s	3h24m22s	2d14h51m58s	9d17h05m47s
twolf	3m26s	23m22s	1h15m44s	2d21h56m13s	9d03h35m12s
parser	3m08s	23m32s	1h21m47s	2d23h50m36s	n/a
apsi	3m33s	23m37s	3h44m49s	3d00h08m42s	8d05h20m38s
swim	2m30s	27m45s	3h04m57s	1d17h35m53s	5d01h14m23s

Table D.4: Elapsed times for running the SPEC CPU 2000 benchmarks on various MIPS simulators. *domori* is time on our reference Pentium D machine. *hershey* is an actual MIPS R12000 system. The pre-compiled SPEC benchmarks from the SESC site are used; some (such as `gzip`) are modified to have shorter run-times, which is why the R12000 runs them faster than the Pentium D.

Benchmark	domori	hershey	qemu_cache	sesc
164.gzip.log	13s	15s	13m52s	2h17m38s
164.gzip.source	29s	25s	24m00s	3h58m48s
164.gzip.program	52s	34s	32m49s	5h27m31s
176.gcc.expr	4s	37s	32m16s	6h12m10s
176.gcc.integrate	5s	39s	34m31s	7h05m16s
164.gzip.random	24s	39s	36m16s	6h01m46s
164.gzip.graphic	29s	47s	49m07s	8h02m23s
256.bzip2.source	40s	2m09s	2h16m34s	1d00h25m02s
176.gcc.166	23s	2m35s	2h03m00s	1d07h14m07s
256.bzip2.program	41s	2m43s	2h58m24s	1d07h32m39s
256.bzip2.graphic	51s	3m07s	3h39m48s	2d00h52m22s
176.gcc.scilab	23s	3m17s	2h47m22s	1d11h13m34s
176.gcc.200	40s	5m39s	6h49m07s	2d15h11m51s
179.art.110	1m15s	10m23s	1h57m46s	2d08h51m01s
179.art.470	1m22s	11m26s	2h06m30s	2d11h09m05s
175.vpr.place	1m28s	13m16s	5h54m56s	2d12h27m59s
183.equake	1m09s	13m41s	6h35m58s	3d17h27m29s
168.wupwise	1m40s	14m55s	21h00m46s	7d08h57m22s
177.mesa	1m36s	14m58s	14h06m14s	5d16h45m08s
254.gap	1m13s	15m36s	11h40m59s	5d00h02m12s
186.crafty	1m12s	16m30s	12h22m53s	4d13h06m39s
200.sixtrack	2m46s	17m38s	1d20h20m47s	10d03h12m14s
175.vpr.route	1m25s	17m38s	6h02m32s	2d02h27m09s
181.mcf	5m09s	28m55s	2h59m49s	2d00h28m10s
197.parser	3m08s	29m05s	17h09m46s	7d23h33m44s
188.ammp	2m47s	29m31s	17h41m38s	7d10h05m58s
173.applu	2m42s	31m24s	20h40m23s	11d04h04m19s
172.mgrid	1m52s	31m59s	1d06h03m47s	16d02h53m15s
300.twolf	3m26s	33m50s	15h13m50s	6d13h20m15s
301.apsi	3m33s	53m35s	1d01h06m40s	10d23h44m24s
171.swim	2m30s	57m23s	11h52m41s	6d12h03m34s

Table D.5: Elapsed times for running the SPEC CPU 2000 benchmarks on various SPARC simulators. *domori* is time on our reference Pentium D machine. *niagara* is an actual SPARC niagara system.

Benchmark	domori	niagara	qemu_bbv
perlbmk.makernd	1s	5s	15s
gcc.expr	4s	25s	1m40s
gcc.integrate	5s	28s	1m40s
perlbmk.perf	13s	1m23s	5m05s
gcc.166	23s	1m40s	5m40s
gzip.log	13s	1m22s	2m27s
gcc.scilab	23s	2m07s	8m39s
art.110	1m15s	9m46s	17m22s
eon.rushmeier	18s	7m52s	17m22s
art.470	1m22s	10m41s	18m08s
gzip.source	29s	2m53s	5m05s
mcf	5m09s	11m35s	9m02s
gcc.200	40s	3m45s	13m22s
gzip.random	24s	2m41s	5m19s
vpr.route	1m25s	10m06s	n/a
eon.cook	22s	2m54s	25m31s
gzip.graphic	29s	3m05s	6m45s
bzip2.source	40s	4m25s	7m42s
bzip2.program	41s	5m06s	8m38s
eon.kajiya	34s	14m44s	39m07s
vpr.place	1m28s	10m00s	n/a
bzip2.graphic	51s	6m02s	9m43s
gzip.program	52s	5m46s	9m07s
equake	1m09s	38m51s	1h16m43s
crafty	1m12s	15m04s	33m39s
gap	1m13s	10m01s	26m26s
swim	2m30s	1h23m48s	2h24m19s
lucas	2m02s	52m04s	2h07m59s
twolf	3m26s	22m41s	23m47s
facerec	3m56s	46m18s	n/a
fma3d	5m31s	1h13m00s	2h43m36s
wupwise	1m40s	1h01m14s	2h37m21s
mesa	1m36s	31m04s	1h06m40s
ammp	2m47s	1h31m38s	3h32m39s
parser	3m08s	n/a	n/a
galgel	1m54s	1h28m53s	2h44m34s
apsi	3m33s	1h25m21s	3h05m47s
applu	2m42s	1h35m48s	3h46m28s
mgrid	1m52s	2h25m51s	5h38m12s
sixtrack	2m46s	2h38m15s	n/a

Table D.6: Times for x86 architecture

Benchmark	32-bit	64-bit	pin	qemu	valgrind	m5
perlbmk.mkrnd	4s	1s	47s	31s	43s	n/a
gcc.expr	5s	5s	2m33s	2m06s	3m15s	5h42m46s
gcc.integrate	5s	5s	2m31s	1m49s	3m05s	5h50m11s
perlbmk.prfct	14s	13s	5m20s	7m49s	10m51s	n/a
gzip.log	17s	13s	4m28s	4m17s	7m43s	20h23m11s
eon.rushmeier	31s	18s	8m06s	21m43s	27m54s	n/a
eon.cook	37s	23s	9m52s	30m24s	38m09s	n/a
gcc.166	12s	23s	5m07s	5m09s	9m08s	18h48m12s
gcc.scilab	20s	24s	10m02s	10m20s	16m48s	1d05h57m27s
gzip.random	28s	24s	8m35s	8m57s	16m22s	1d20h32m49s
gzip.graphic	35s	30s	10m00s	11m24s	19m54s	2d06h59m35s
gzip.source	28s	31s	9m47s	8m27s	15m20s	1d16h01m59s
eon.kajiya	54s	35s	13m09s	39m22s	50m49s	n/a
bzip2.program	48s	41s	13m00s	15m38s	26m50s	3d01h45m46s
gcc.200	34s	41s	14m30s	16m55s	28m09s	2d05h54m01s
bzip2.source	47s	42s	11m40s	13m35s	23m03s	2d15h18m22s
bzip2.graphic	59s	53s	15m12s	17m47s	30m04s	3d12h43m40s
gzip.program	53s	53s	17m22s	15m31s	29m57s	3d03h46m29s
equake	1m27s	1m11s	7m59s	35m55s	41m24s	n/a
crafty	1m42s	1m12s	26m41s	1h16m48s	1h44m35s	5d21h50m20s
art.110	1m27s	1m12s	9m55s	27m02s	18m15s	n/a
gap	1m33s	1m14s	33m21s	1h02m41s	1h20m34s	6d10h04m29s
art.470	1m52s	1m16s	10m38s	27m27s	20m08s	n/a
vpr.route	1m39s	1m26s	10m53s	20m06s	29m12s	n/a
vpr.place	1m07s	1m30s	14m27s	30m27s	48m03s	n/a
wupwise	3m20s	1m46s	43m12s	2h29m04s	2h51m27s	n/a
galgel	2m23s	1m58s	30m15s	1h01m42s	1h43m08s	n/a
mesa	3m13s	1m58s	24m36s	1h45m27s	2h51m15s	n/a
lucas	4m50s	2m09s	17m09s	1h34m40s	1h23m06s	n/a
ammp	4m35s	2m52s	29m43s	1h50m27s	1h52m03s	n/a
parser	3m21s	3m18s	56m41s	1h33m21s	1h55m06s	11d07h24m21s
applu	8m18s	3m21s	28m18s	2h53m12s	3h12m13s	n/a
twolf	3m15s	3m28s	34m17s	1h57m09s	2h09m56s	n/a
mgrid	3m46s	3m49s	15m27s	1h53m41s	2h06m23s	n/a
swim	3m18s	3m51s	10m16s	1h16m52s	1h19m14s	n/a
facerec	3m53s	4m10s	24m11s	1h11m21s	1h30m02s	n/a
apsi	4m28s	4m47s	30m16s	2h38m45s	3h06m16s	n/a
mcf	2m45s	5m24s	14m22s	14m32s	22m01s	n/a
fma3d	5m21s	5m35s	33m35s	2h04m33s	2h25m38s	n/a
sixtrack	4m40s	18m24s	27m08s	4h02m57s	4h21m14s	n/a

Table D.7: Times for x86_64 architecture comparing simulators.

Benchmark	domori	m5-nocache	m5-cache	ptlsim
perlbmk.makerand	1s	53m48s	58m52s	n/a
gcc.expr	5s	4h53m12s	5h33m00s	10h30m07s
gcc.integrate	5s	4h50m33s	5h52m55s	10h16m38s
perlbmk.perfect	13s	14h32m05s	15h53m58s	n/a
gzip.log	13s	19h39m50s	20h24m41s	1d14h58m40s
eon.rushmeier	18s	1d09h55m28s	1d07h58m31s	2d17h09m28s
eon.cook	23s	1d19h02m39s	1d16h51m53s	3d09h41m21s
gcc.166	23s	16h37m58s	20h38m33s	1d09h50m05s
gcc.scilab	24s	1d01h55m48s	1d05h23m15s	2d08h30m16s
gzip.random	24s	1d13h24m45s	1d15h12m09s	3d08h57m53s
gzip.graphic	30s	2d01h39m03s	2d03h34m21s	4d18h23m40s
gzip.source	31s	1d21h33m05s	1d23h23m02s	3d21h11m05s
eon.kajiya	35s	2d09h24m17s	2d07h21m29s	5d07h00m46s
bzip2.program	41s	2d15h17m39s	2d19h00m34s	6d00h17m33s
gcc.200	41s	1d21h23m22s	2d07h14m04s	4d02h12m45s
bzip2.source	42s	2d03h19m57s	2d06h53m50s	5d03h11m06s
bzip2.graphic	53s	3d01h38m12s	3d05h03m02s	6d16h48m50s
gzip.program	53s	4d01h02m04s	4d05h04m40s	7d21h37m40s
equake	1m11s	2d22h59m11s	2d14h41m01s	7d16h12m56s
crafty	1m12s	4d04h15m47s	4d08h47m26s	9d01h36m55s
art.110	1m12s	21h25m23s	1d06h28m02s	2d05h29m40s
gap	1m14s	5d07h28m52s	5d15h42m02s	11d14h07m02s
art.470	1m16s	1d04h40m49s	1d09h41m38s	2d11h07m00s
vpr.route	1m26s	1d21h55m53s	2d03h26m07s	n/a
vpr.place	1m30s	2d16h46m57s	2d21h32m58s	n/a
wupwise	1m46s	8d07h15m28s	8d17h10m31s	17d15h57m30s
galgel	1m58s	6d16h04m16s	7d16h52m55s	18d08h38m56s
mesa	1m58s	7d02h31m53s	6d15h05m59s	12d10h17m04s
lucas	2m09s	6d03h01m15s	5d12h26m48s	12d16h59m44s
ammp	2m52s	6d19h42m04s	7d23h34m04s	18d00h55m59s
parser	3m18s	8d03h55m04s	8d18h34m43s	18d04h31m22s
applu	3m21s	8d07h21m55s	9d09h47m46s	20d17h07m13s
twolf	3m28s	8d07h34m14s	9d03h33m07s	17d21h00m02s
mgrid	3m49s	7d20h02m09s	9d00h43m12s	19d08h28m21s
swim	3m51s	4d18h36m35s	5d19h04m44s	13d06h49m15s
facerec	4m10s	6d08h53m58s	7d00h03m23s	n/a
apsi	4m47s	9d08h01m41s	9d05h47m29s	20d11h39m32s
mcf	5m24s	1d08h17m02s	3d11h32m14s	4d00h18m09s
fma3d	5m35s	7d23h11m24s	8d02h44m38s	17d23h13m33s
sixtrack	18m24s	11d14h34m06s	10d18h02m56s	n/a

Table D.8: Times for x86_64 DBI

Benchmark	domori	pin	qemu	valgrind
perlbmk.makerand	1s	2s	n/a	6s
gcc.integrate	5s	17s	1m21s	33s
gcc.expr	4s	18s	1m03s	38s
perlbmk.perfect	13s	53s	n/a	2m30s
gzip.log	13s	15s	2m39s	2m56s
eon.rushmeier	18s	37s	5m47s	5m23s
eon.cook	22s	41s	7m33s	7m27s
gcc.166	23s	40s	1m51s	1m33s
gcc.scilab	23s	52s	3m42s	3m18s
gzip.random	24s	32s	4m47s	5m34s
gzip.source	29s	48s	5m58s	2m23s
gzip.graphic	29s	56s	7m12s	2m56s
eon.kajiya	34s	1m09s	10m16s	9m53s
gcc.200	40s	1m17s	6m27s	5m17s
bzip2.source	40s	53s	7m43s	3m35s
bzip2.program	41s	1m02s	8m51s	4m01s
bzip2.graphic	51s	1m15s	9m58s	4m35s
gzip.program	52s	1m18s	12m01s	4m36s
equake	1m09s	1m20s	8m09s	4m15s
crafty	1m12s	3m00s	17m23s	15m12s
gap	1m13s	2m22s	18m29s	11m25s
art.110	1m15s	1m31s	4m28s	4m02s
art.470	1m22s	1m43s	4m50s	4m09s
vpr.route	1m25s	1m36s	7m42s	6m33s
vpr.place	1m28s	1m39s	10m03s	23m35s
mesa	1m36s	2m09s	26m40s	1h07m54s
wupwise	1m40s	2m25s	25m10s	15m04s
mgrid	1m52s	1m55s	30m53s	13m21s
galgel	1m54s	2m02s	17m31s	10m21s
lucas	2m02s	2m42s	19m51s	25m37s
swim	2m30s	2m47s	17m47s	9m51s
applu	2m42s	2m51s	25m29s	13m40s
sixtrack	2m46s	2m28s	36m30s	15m26s
ammp	2m47s	2m55s	24m35s	12m52s
parser	3m08s	4m51s	24m56s	15m03s
twolf	3m26s	4m13s	32m42s	27m35s
apsi	3m33s	4m19s	26m03s	16m30s
facerec	3m56s	3m56s	17m14s	19m48s
mcf	5m09s	5m41s	6m46s	8m28s
fma3d	5m31s	5m55s	35m42s	16m08s

Table D.9: Times for x86_64 DBI utilities running cache simulations.

Benchmark	domori	pin_cache	qemu_cache	valgrind_cache
perlbmk.makerand	1s	1m56s	n/a	51s
gcc.expr	5s	11m20s	32m00s	3m56s
gcc.integrate	5s	11m53s	39m38s	3m41s
perlbmk.perfect	13s	35m38s	n/a	21m11s
gzip.log	13s	26m21s	1h12m24s	9m27s
eon.rushmeier	18s	1h05m32s	3d00h03m24s	23m54s
eon.cook	23s	1h25m14s	4d18h35m47s	31m36s
gcc.166	23s	42m49s	3h08m00s	13m08s
gcc.scilab	24s	58m32s	2h56m21s	20m45s
gzip.random	24s	54m41s	2h16m35s	16m57s
gzip.graphic	30s	1h11m05s	2h56m45s	22m54s
gzip.source	31s	1h00m47s	2h43m03s	21m19s
eon.kajiya	35s	1h55m44s	5d17h29m44s	43m32s
bzip2.program	41s	1h34m03s	4h21m22s	31m45s
gcc.200	41s	1h40m16s	5h48m13s	34m47s
bzip2.source	42s	1h19m10s	3h43m54s	27m01s
bzip2.graphic	53s	1h51m39s	11h35m49s	36m09s
gzip.program	53s	2h00m24s	5h43m41s	42m57s
equake	1m11s	2h01m17s	n/a	35m11s
crafty	1m12s	3h35m02s	6h32m49s	1h28m35s
art.110	1m12s	1h10m26s	1d21h39m24s	19m02s
gap	1m14s	4h54m09s	17h30m24s	1h33m39s
art.470	1m16s	1h17m53s	2d00h29m12s	20m47s
vpr.route	1m26s	1h44m57s	4h00m26s	30m12s
vpr.place	1m30s	2h52m41s	17h15m57s	1h02m51s
wupwise	1m46s	10h23m02s	n/a	2h12m48s
galgel	1m58s	5h55m11s	n/a	1h24m49s
mesa	1m58s	5h38m14s	15h38m15s	2h54m54s
lucas	2m09s	4h33m53s	n/a	1h20m45s
ammp	2m52s	6h26m20s	n/a	1h48m44s
parser	3m18s	6h16m05s	18h18m15s	1h59m23s
applu	3m21s	7h52m02s	n/a	2h01m57s
twolf	3m28s	10h54m13s	23h36m00s	2h37m57s
mgrid	3m49s	7h26m15s	n/a	1h45m15s
swim	3m51s	6h06m45s	n/a	1h05m14s
facerec	4m10s	5h06m00s	10h59m14s	1h31m52s
apsi	4m47s	7h55m53s	n/a	2h09m19s
mcf	5m24s	1h17m15s	4h32m23s	30m55s
fma3d	5m35s	7h03m39s	n/a	2h15m55s
sixtrack	18m24s	12h19m14s	n/a	2h24m07s

APPENDIX E

CPI PHASE PLOTS

Program phases can vary even among implementations of the same architecture. Presented here are phase plots for the CPI metric on x86 and x86_64 machines.

E.1 32-bit x86

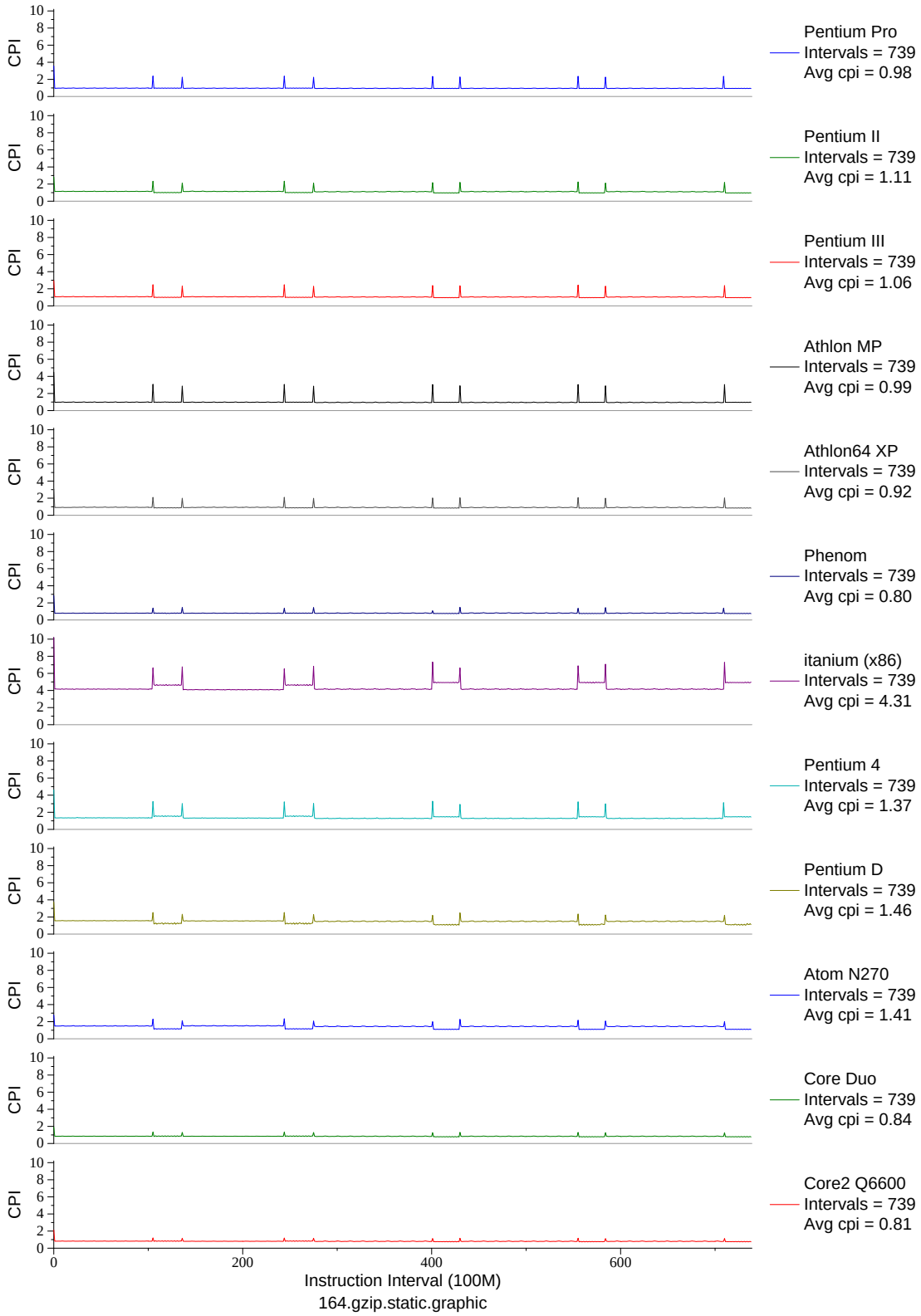

Figure E.1: CPI phase plot for `gzip.graph` (INT, C, Compression)

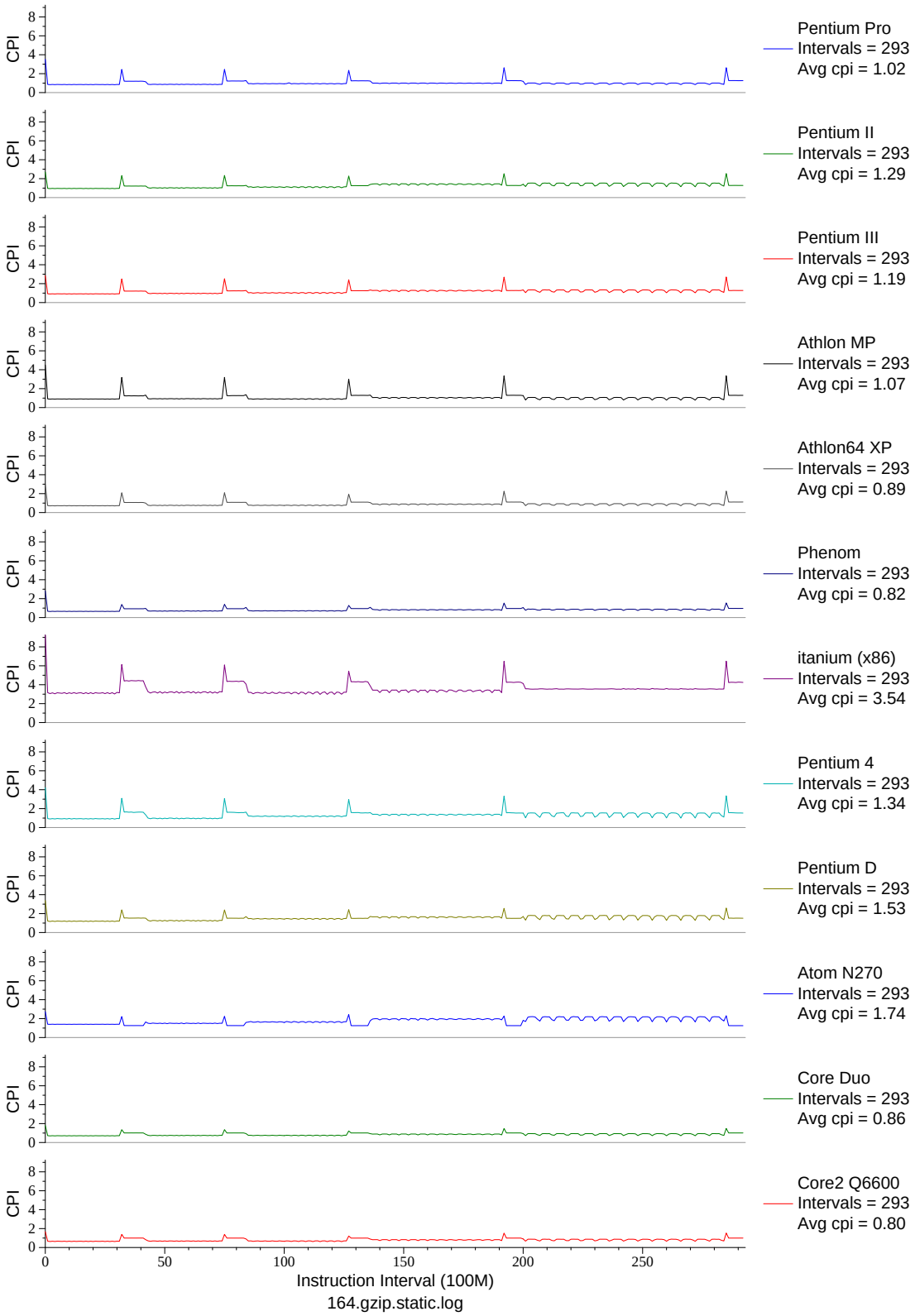

Figure E.2: CPI phase plot for `gzip.log` (INT, C, Compression)

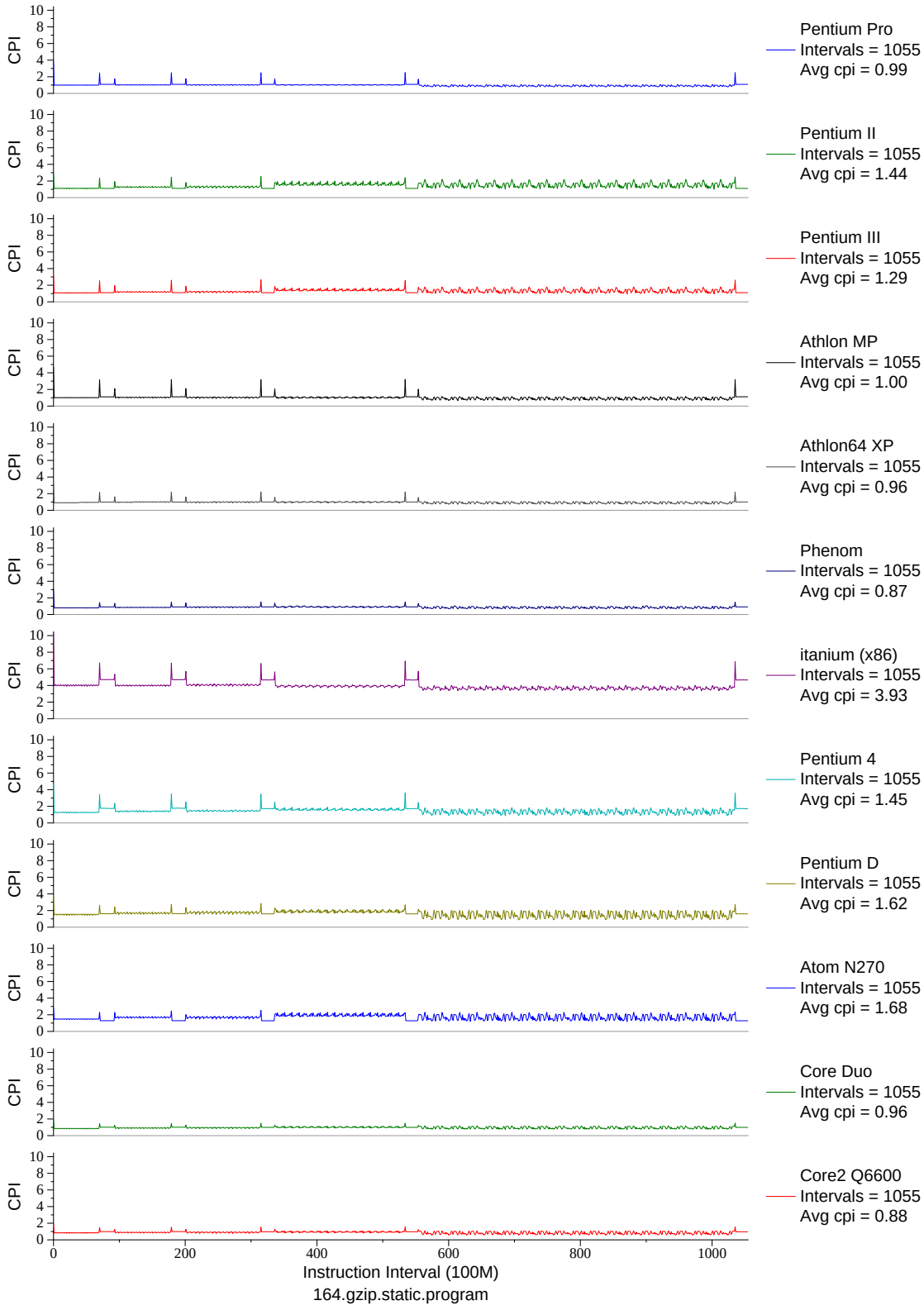

Figure E.3: CPI phase plot for `gzip.prog` (INT, C, Compression)

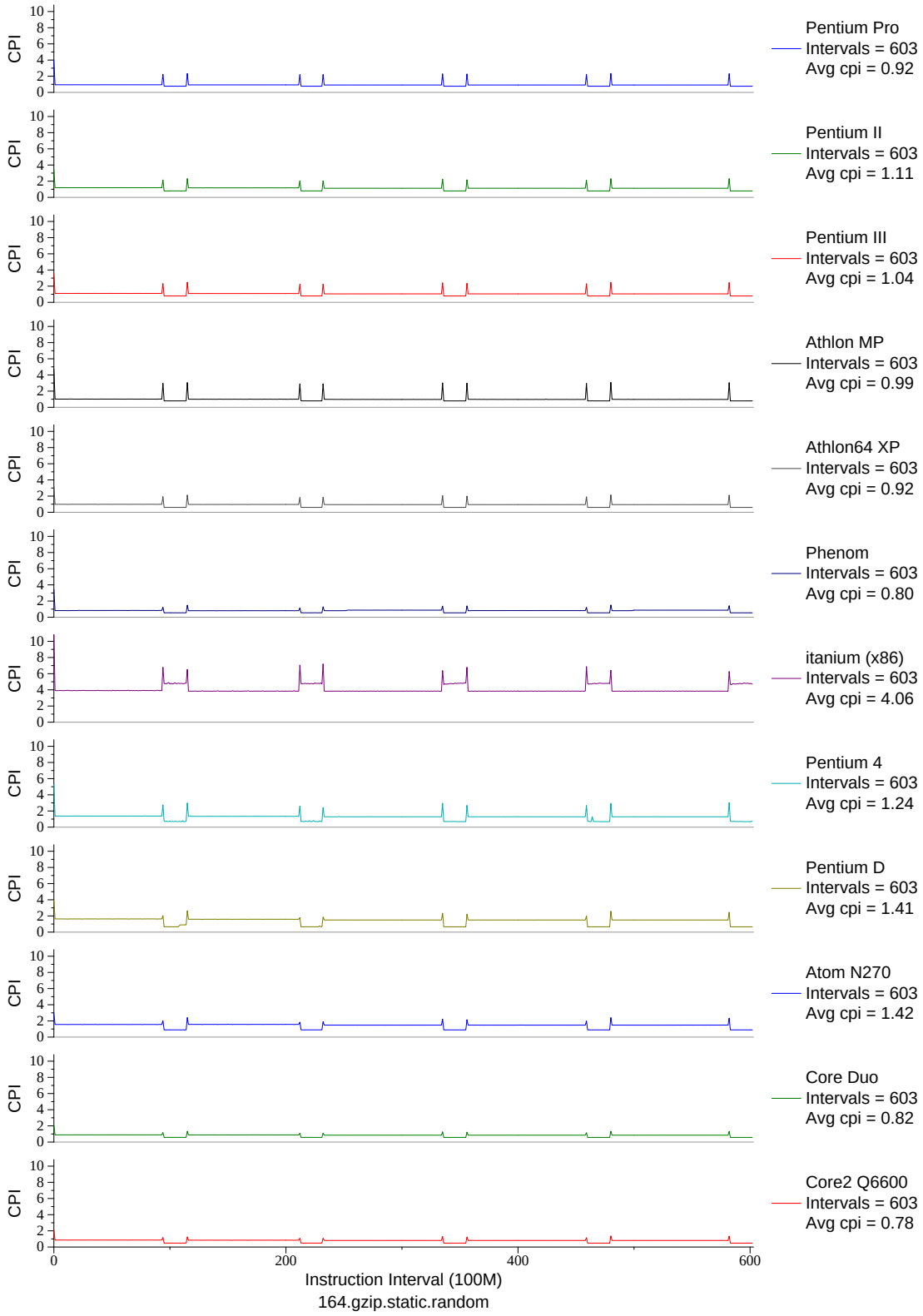

Figure E.4: CPI phase plot for `gzip.rand` (INT, C, Compression)

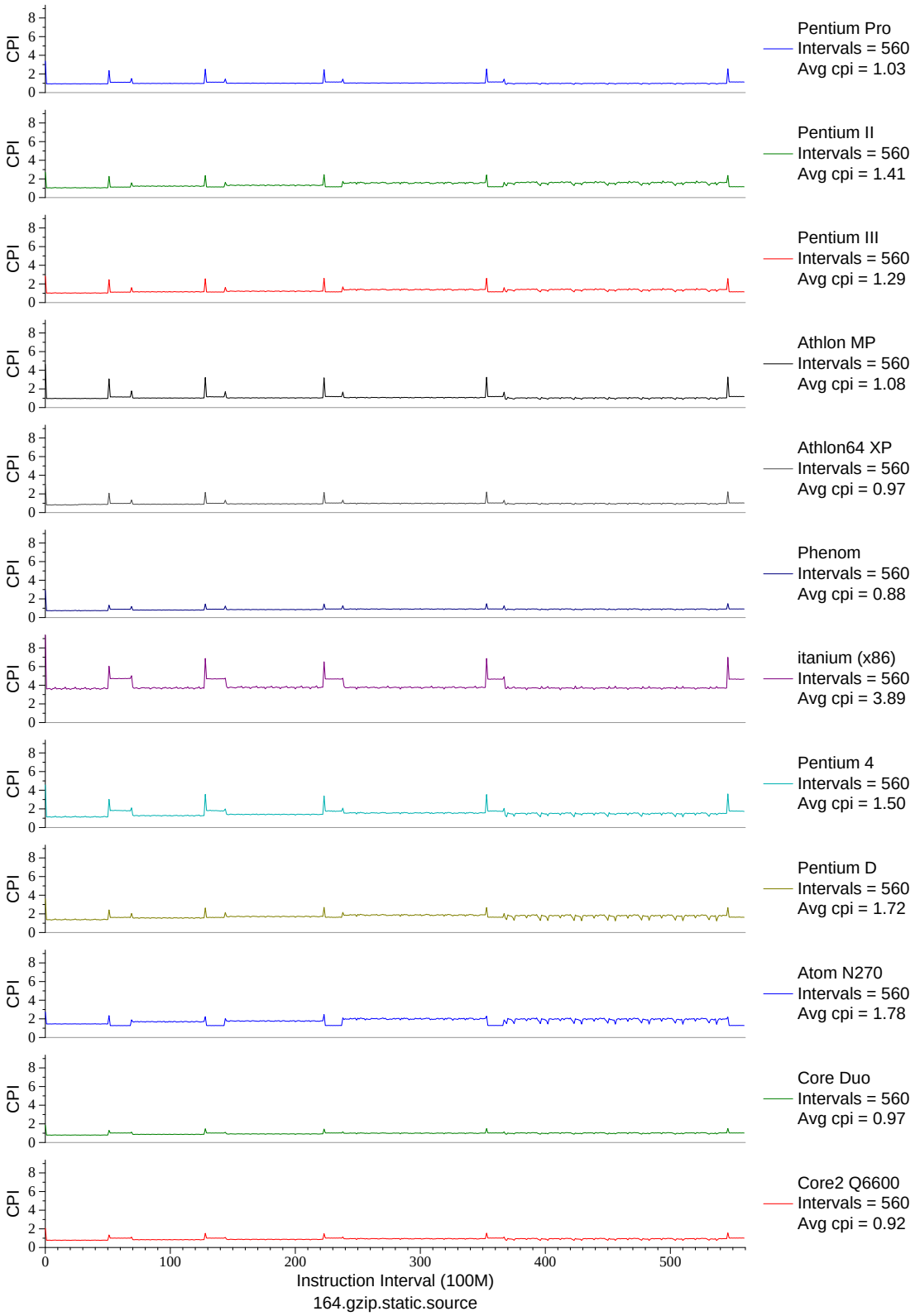

Figure E.5: CPI phase plot for `gzip.src` (INT, C, Compression)

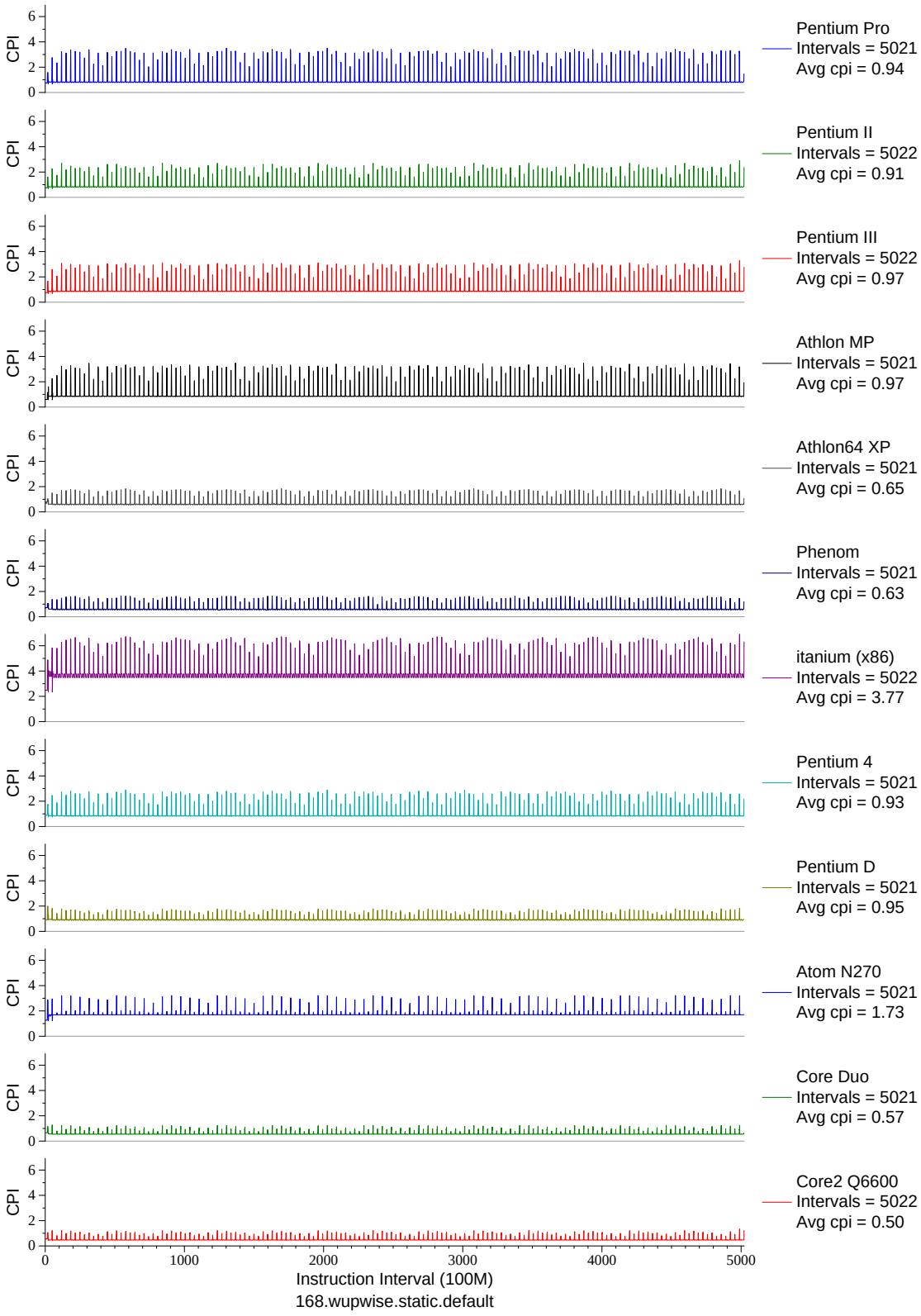

Figure E.6: CPI phase plot for `wupwise` (FP, F77, Quantum Chromodynamics)

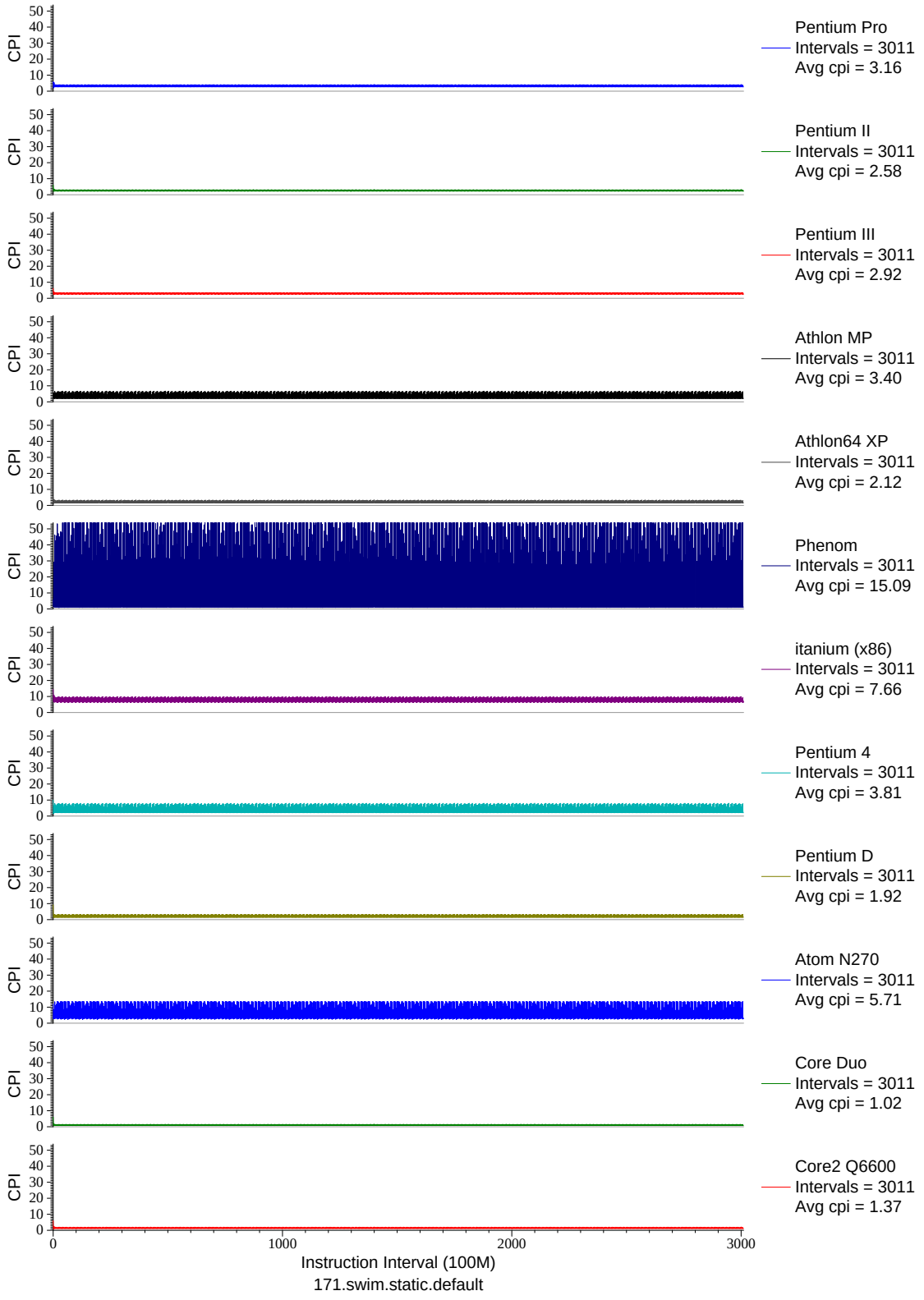

Figure E.7: CPI phase plot for `swim` (FP, F77, Meteorology/Water)

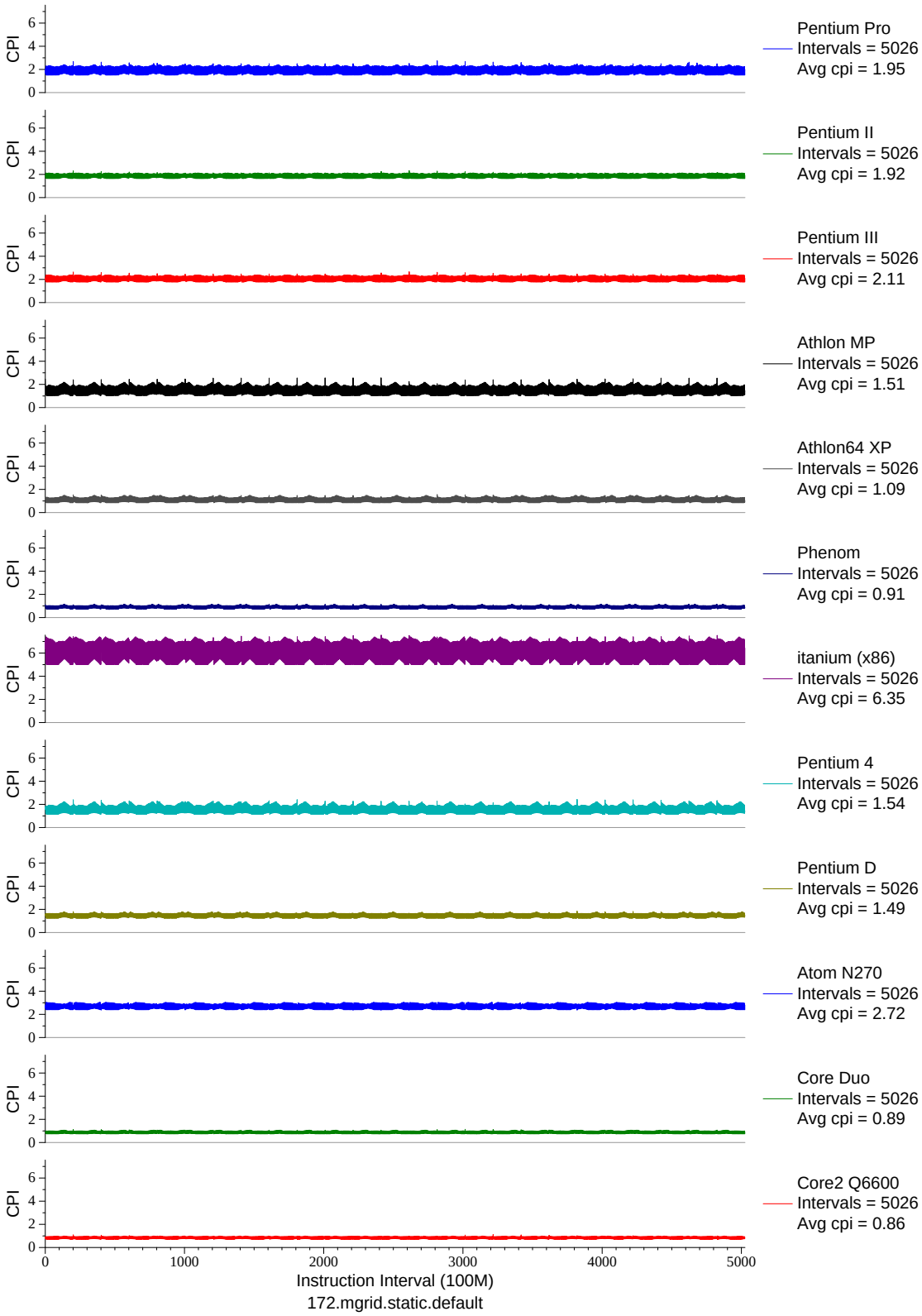

Figure E.8: CPI phase plot for `mgrid` (FP, F77, Multi-Grid Solver)

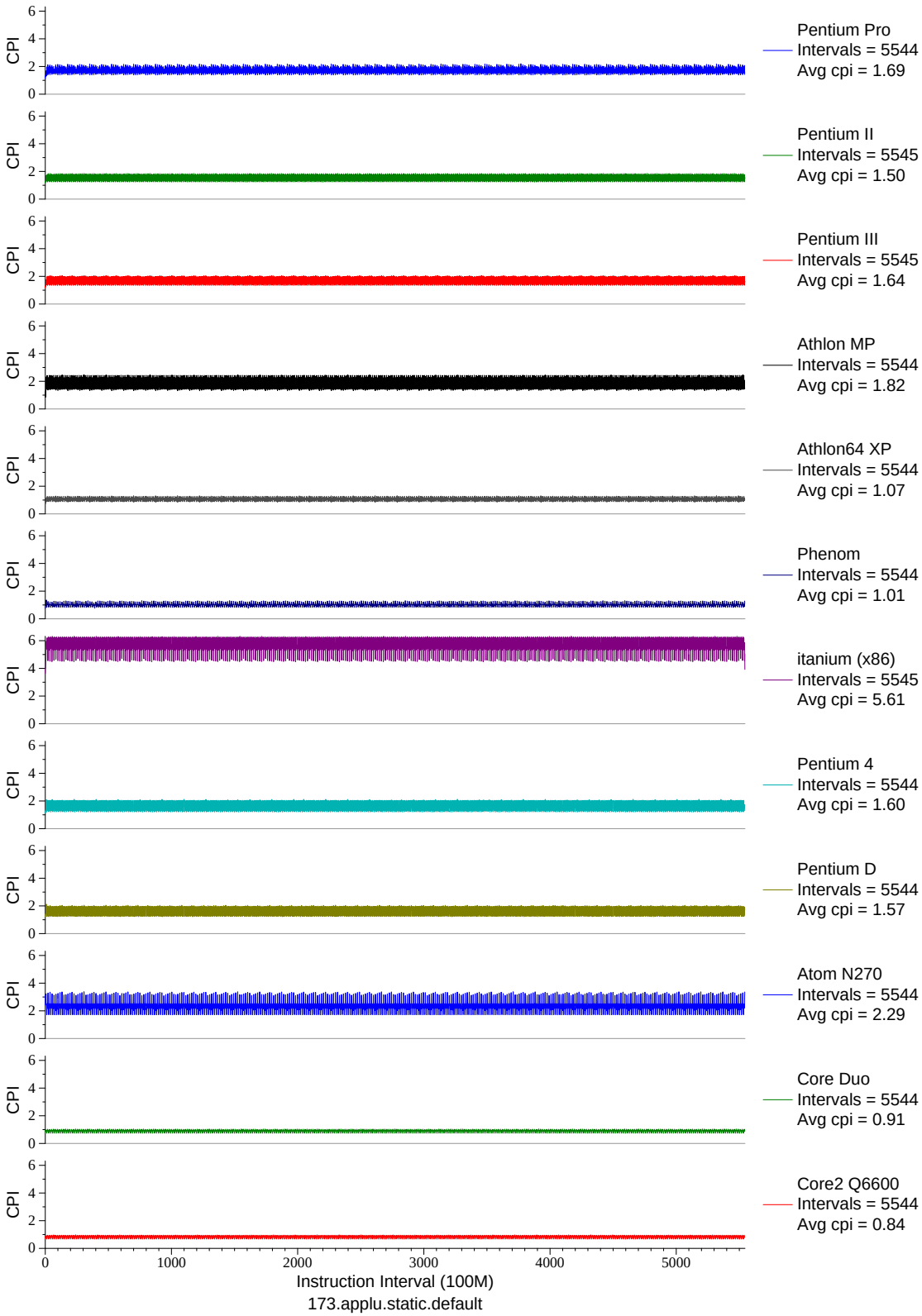

Figure E.9: CPI phase plot for `applu` (FP, F77, Fluid Dynamics)

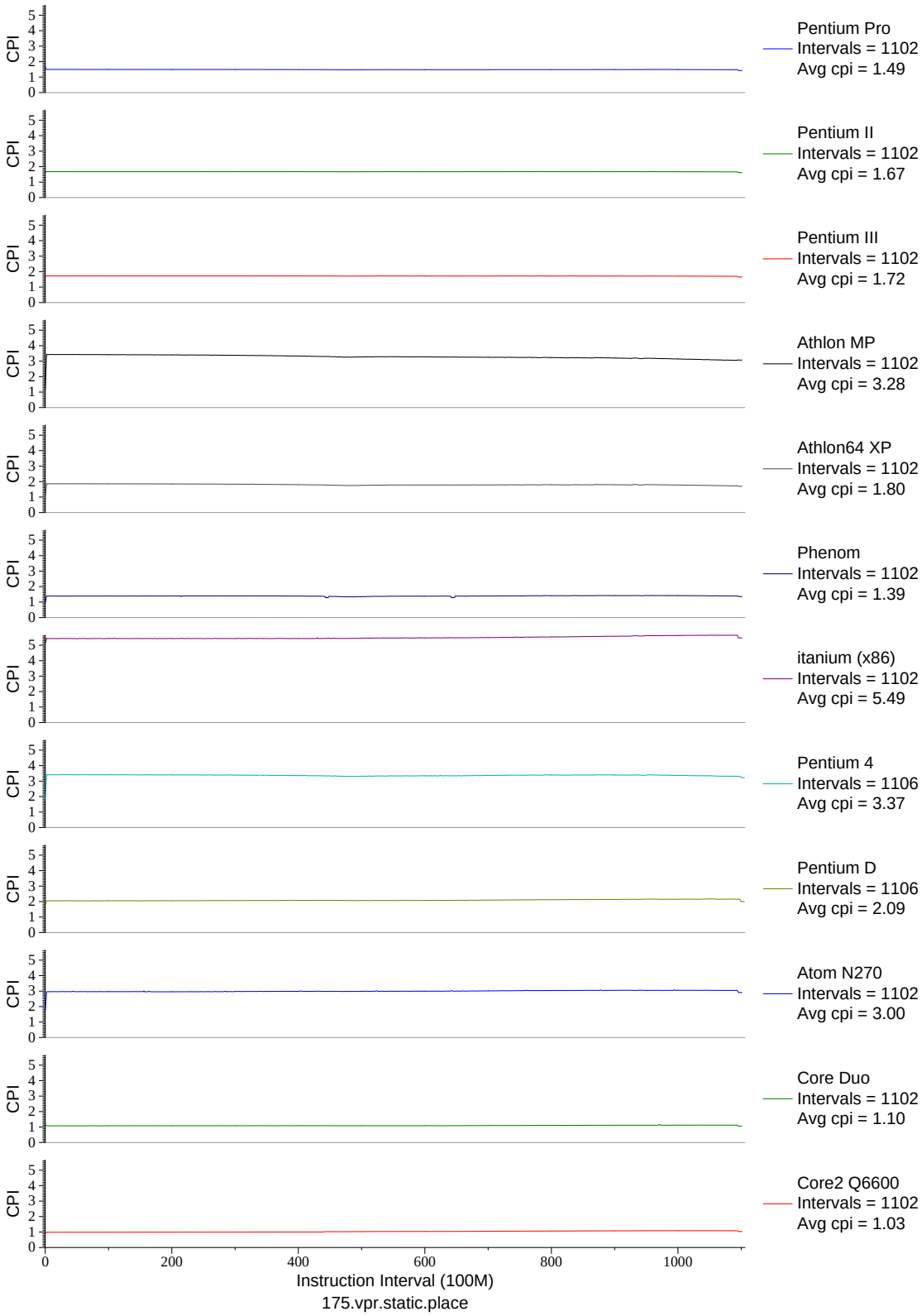

Figure E.10: CPI phase plot for `vpr.place` (INT, C, FPGA Place/Route)

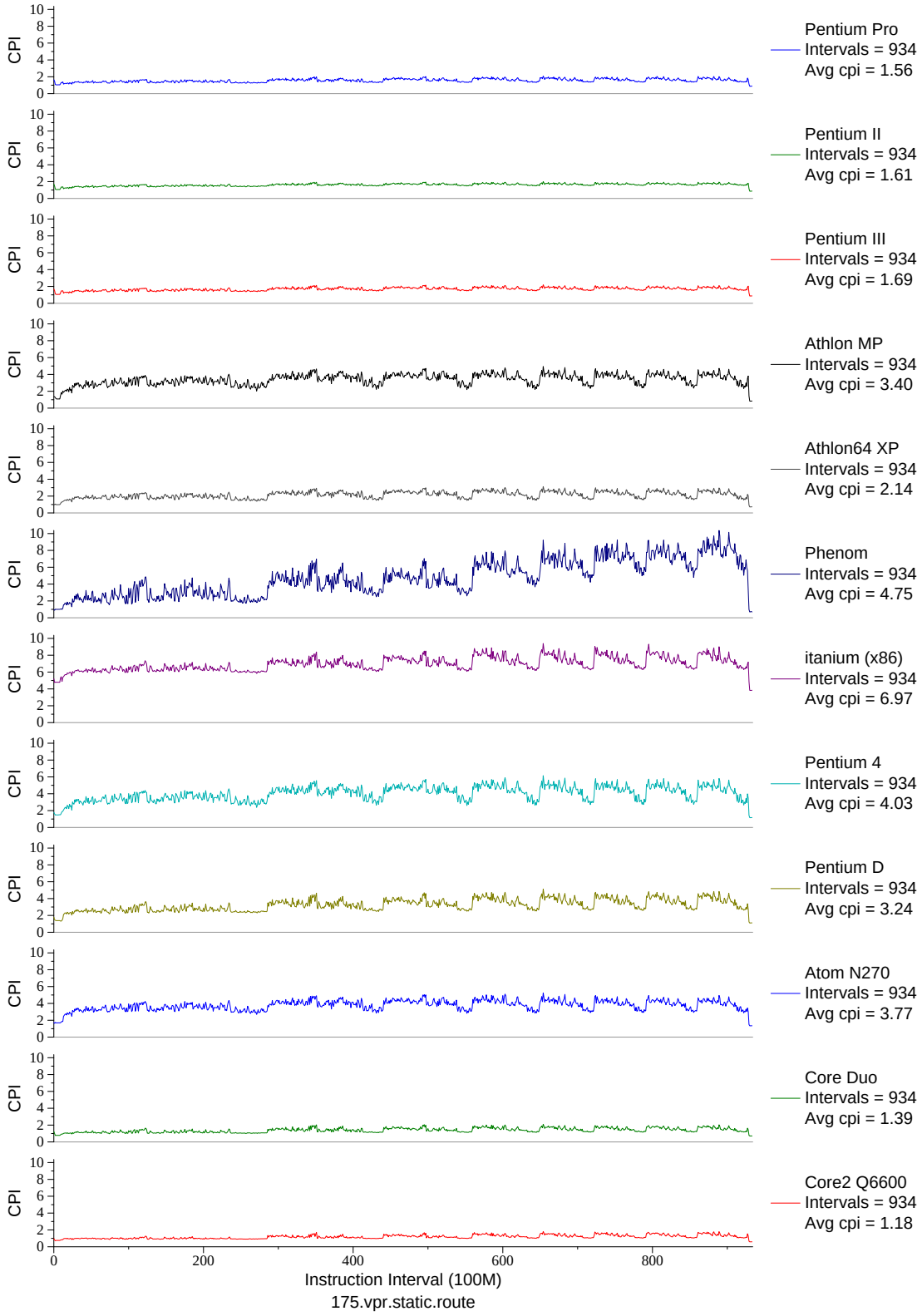

Figure E.11: CPI phase plot for `vpr.route` (INT, C, FPGA Place/Route)

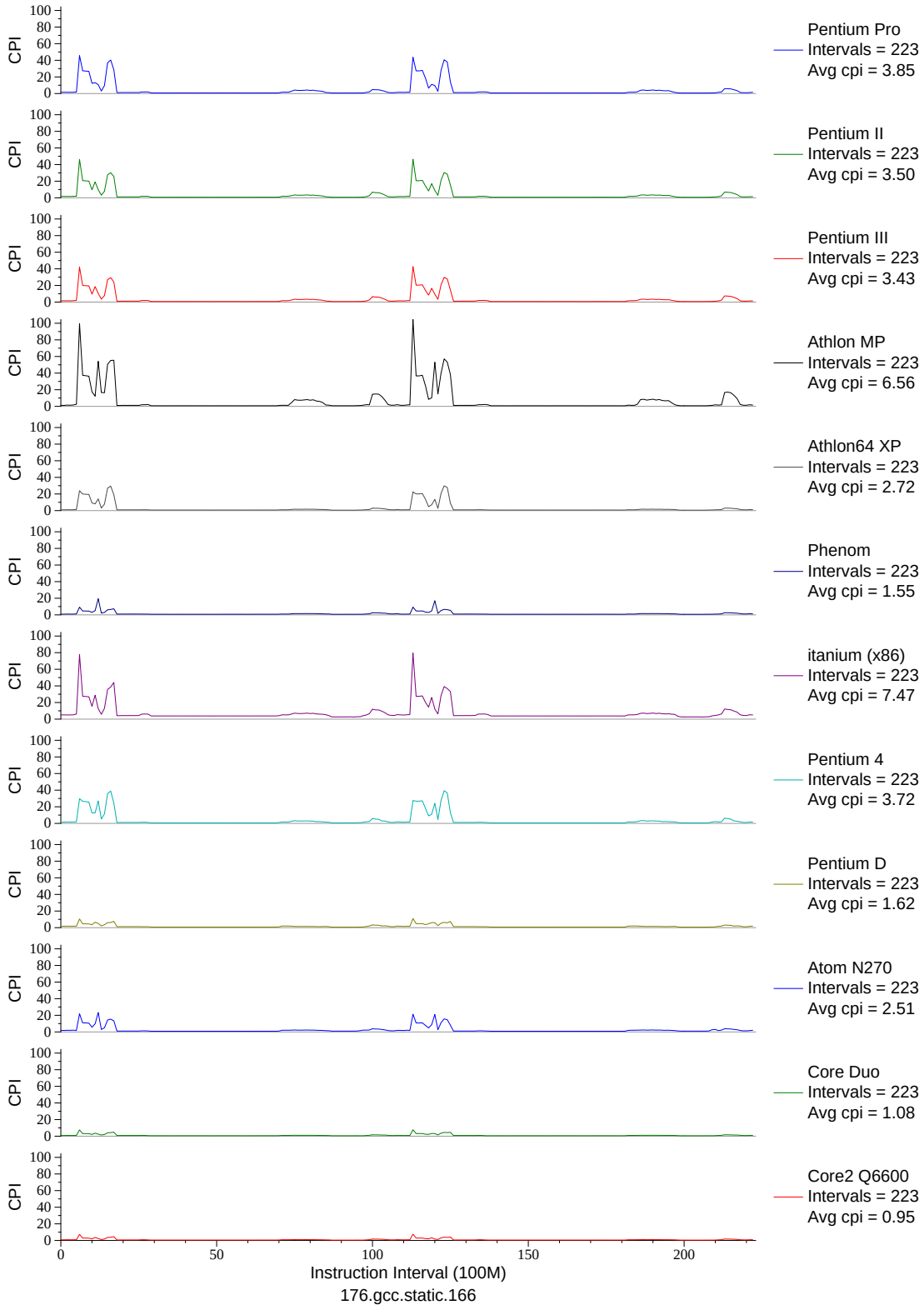

Figure E.12: CPI phase plot for `gcc.166` (INT, C, C Compiler)

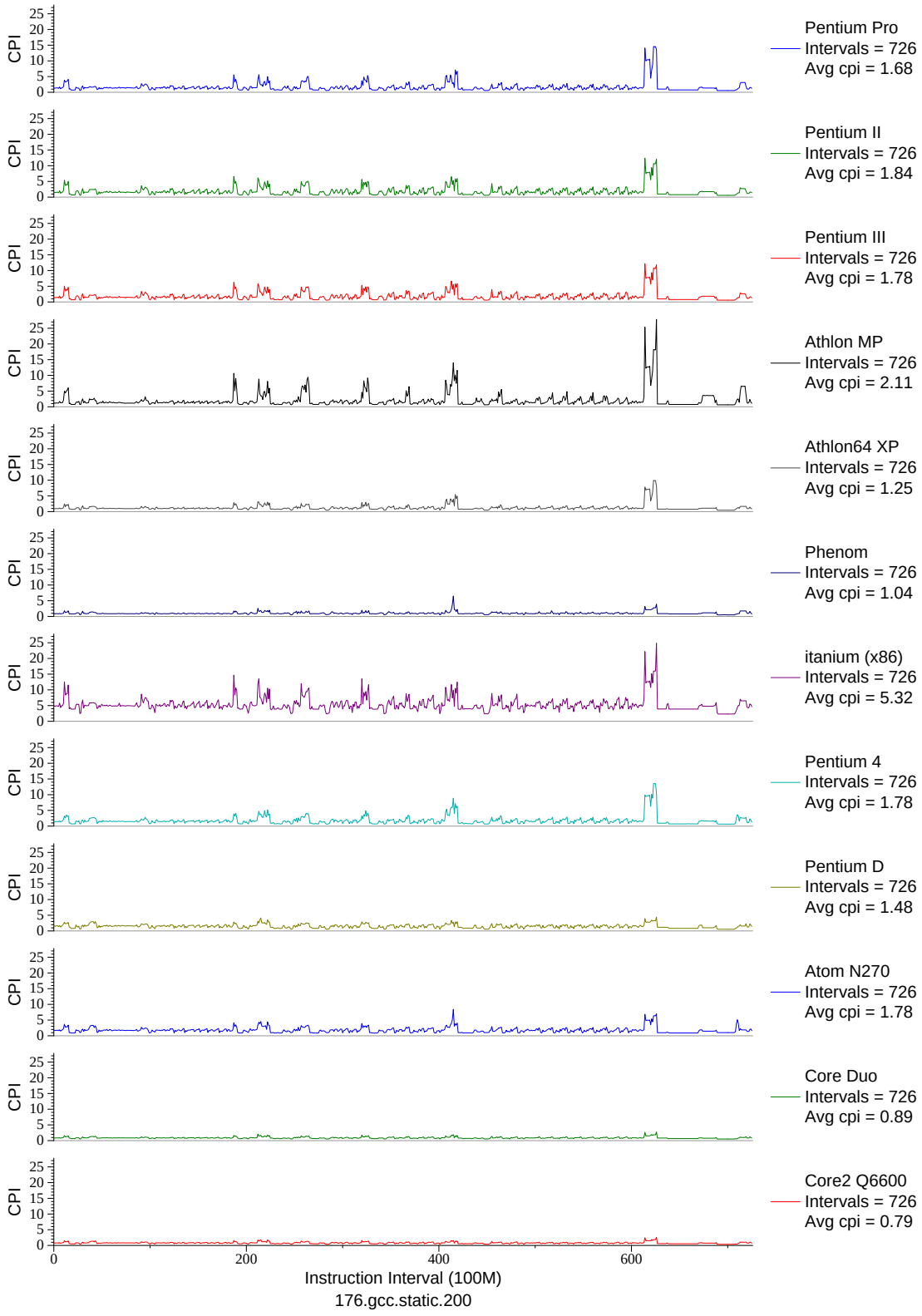

Figure E.13: CPI phase plot for `gcc.200` (INT, C, C Compiler)

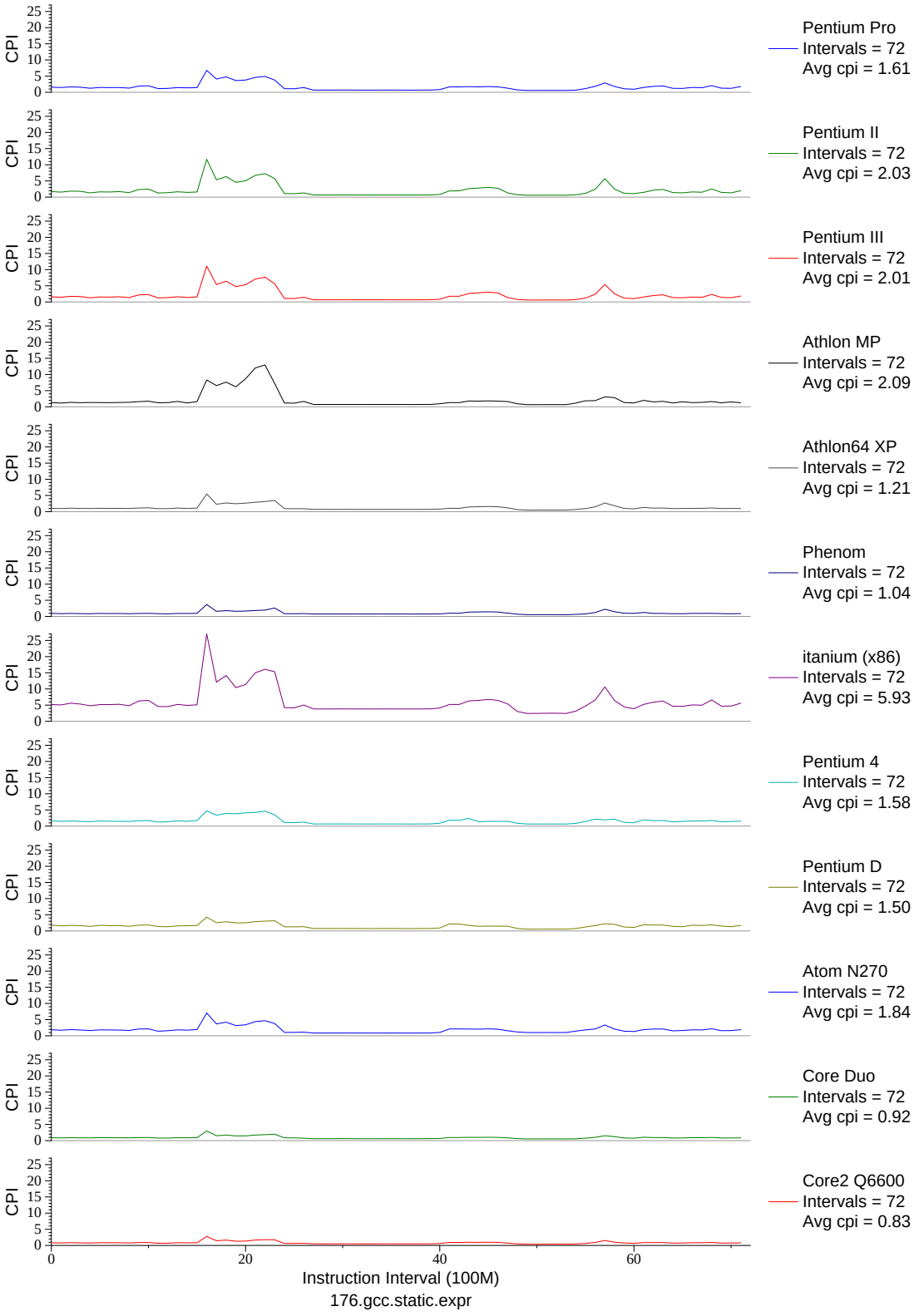

Figure E.14: CPI phase plot for `gcc.expr` (INT, C, C Compiler)

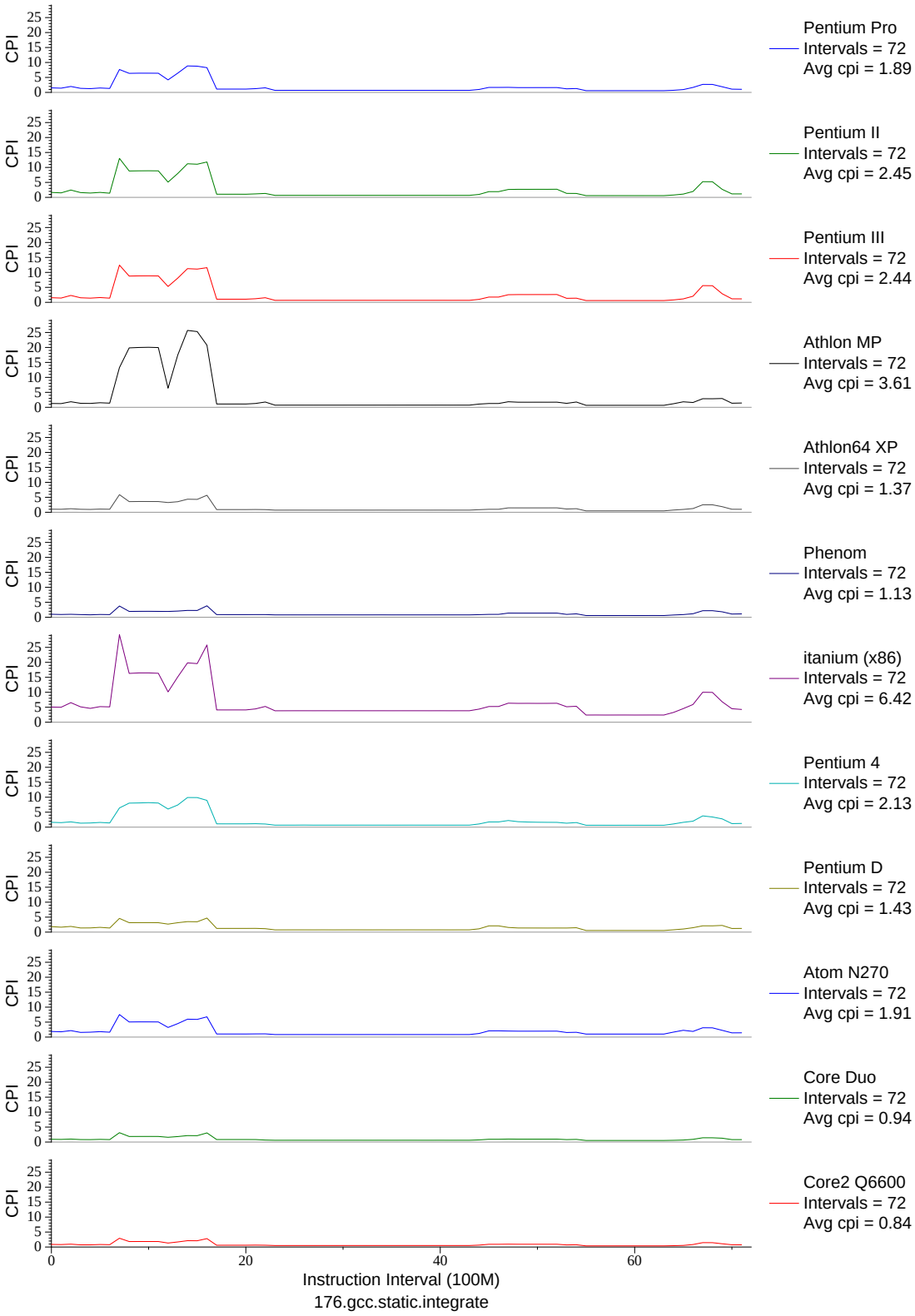

Figure E.15: CPI phase plot for `gcc.int` (INT, C, C Compiler)

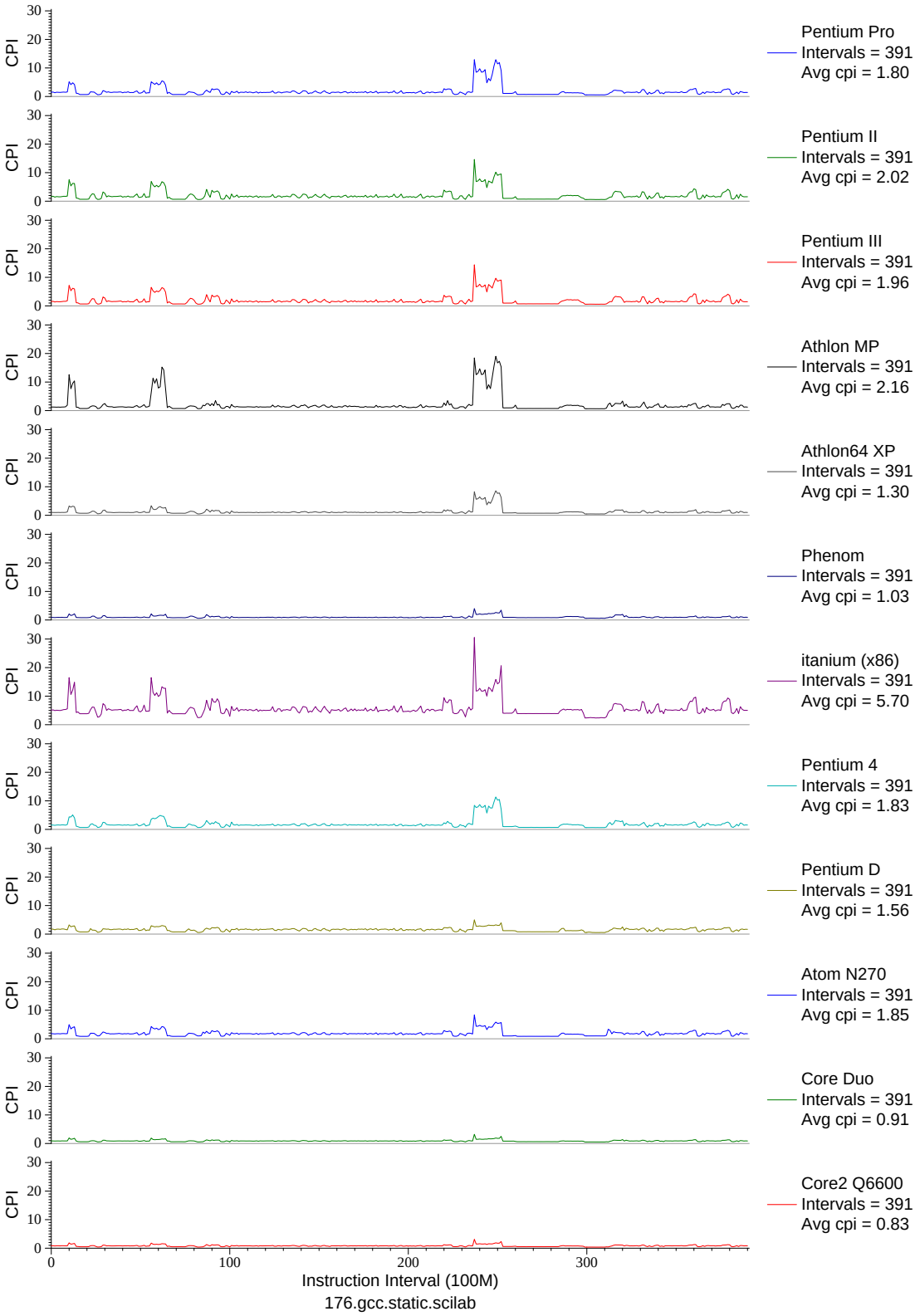

Figure E.16: CPI phase plot for `gcc.sci` (INT, C, C Compiler)

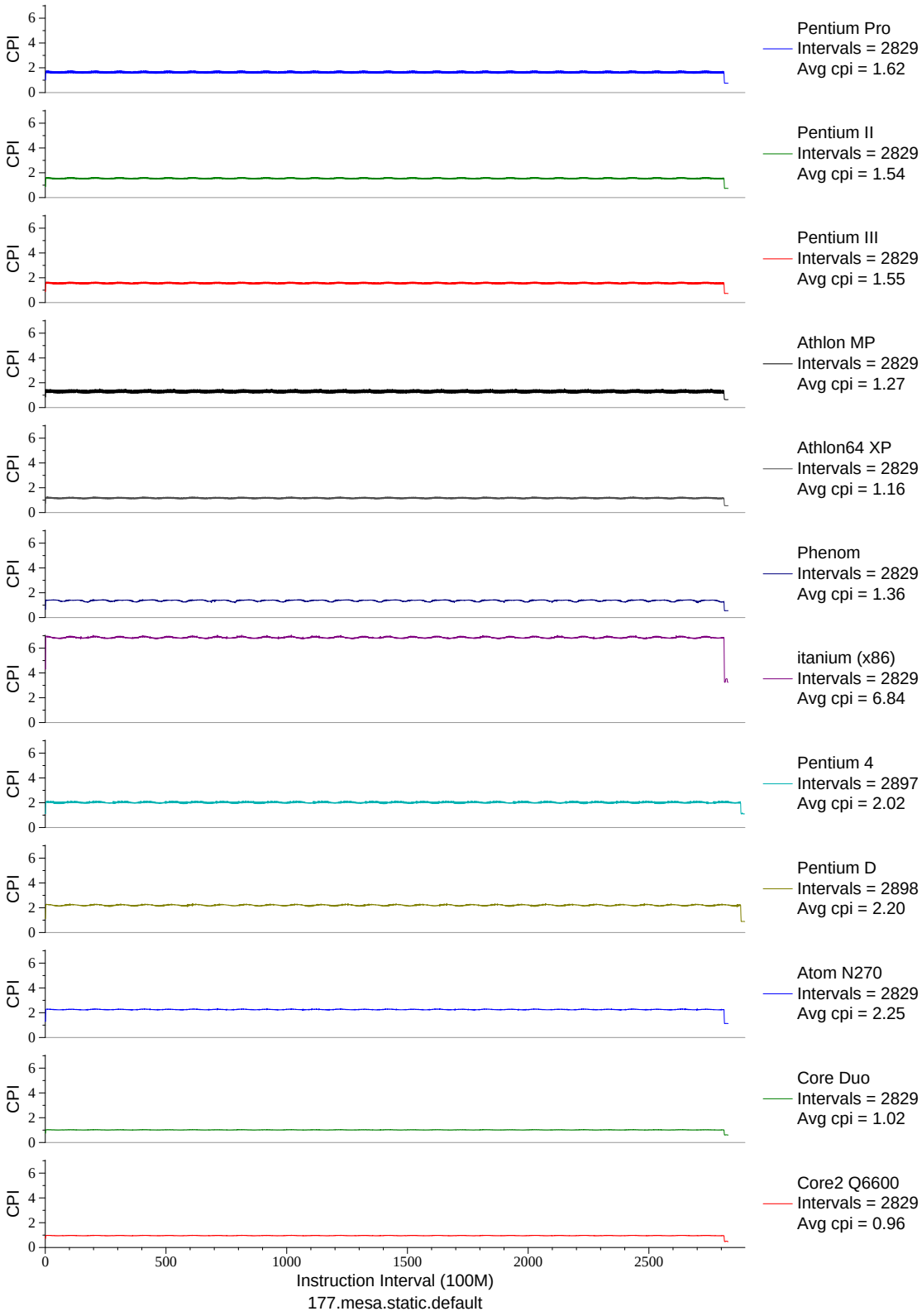

Figure E.17: CPI phase plot for `mesa` (FP, C, 3D-graphics)

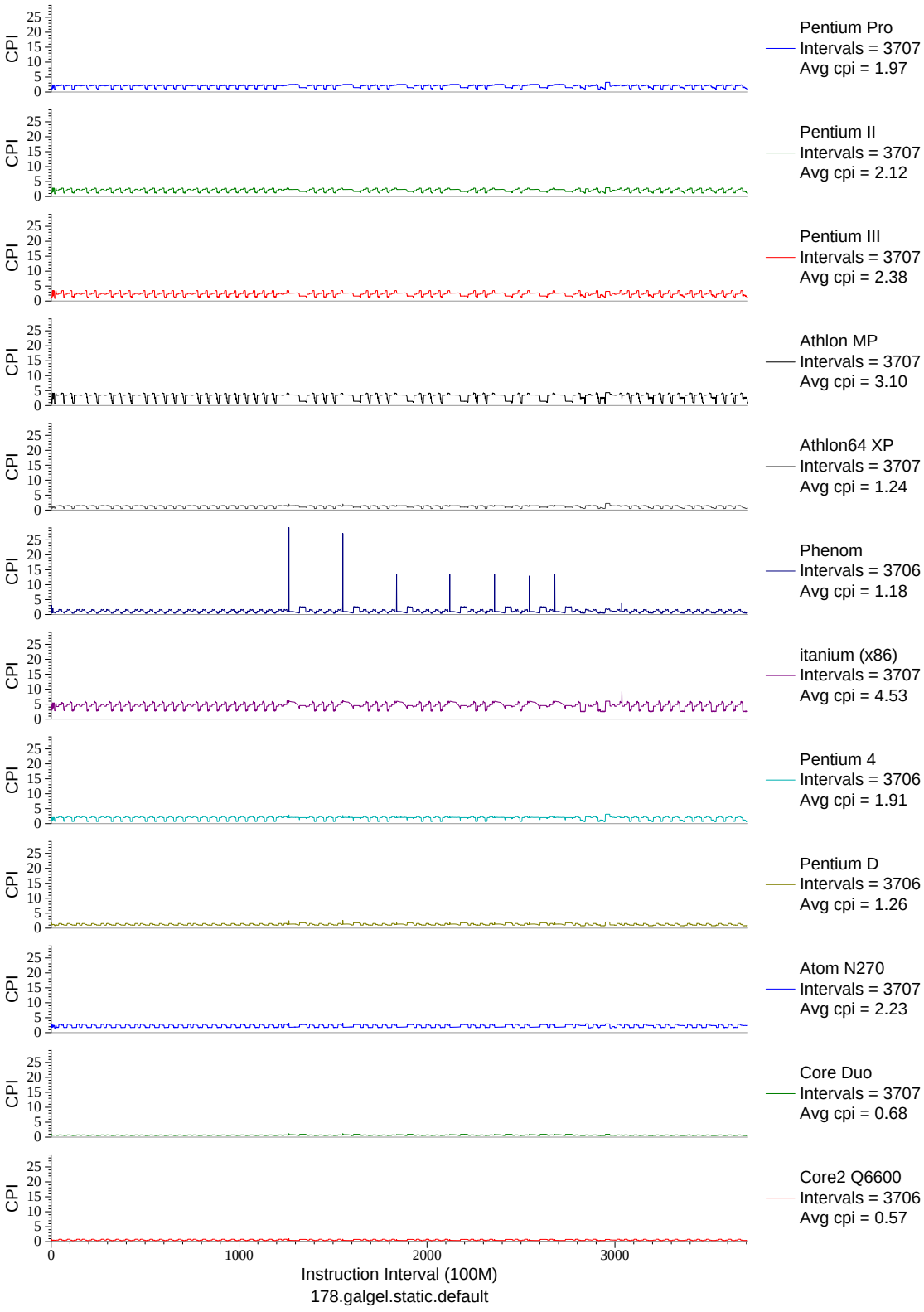

Figure E.18: CPI phase plot for `galgel` (FP, F90, Fluid Dynamics)

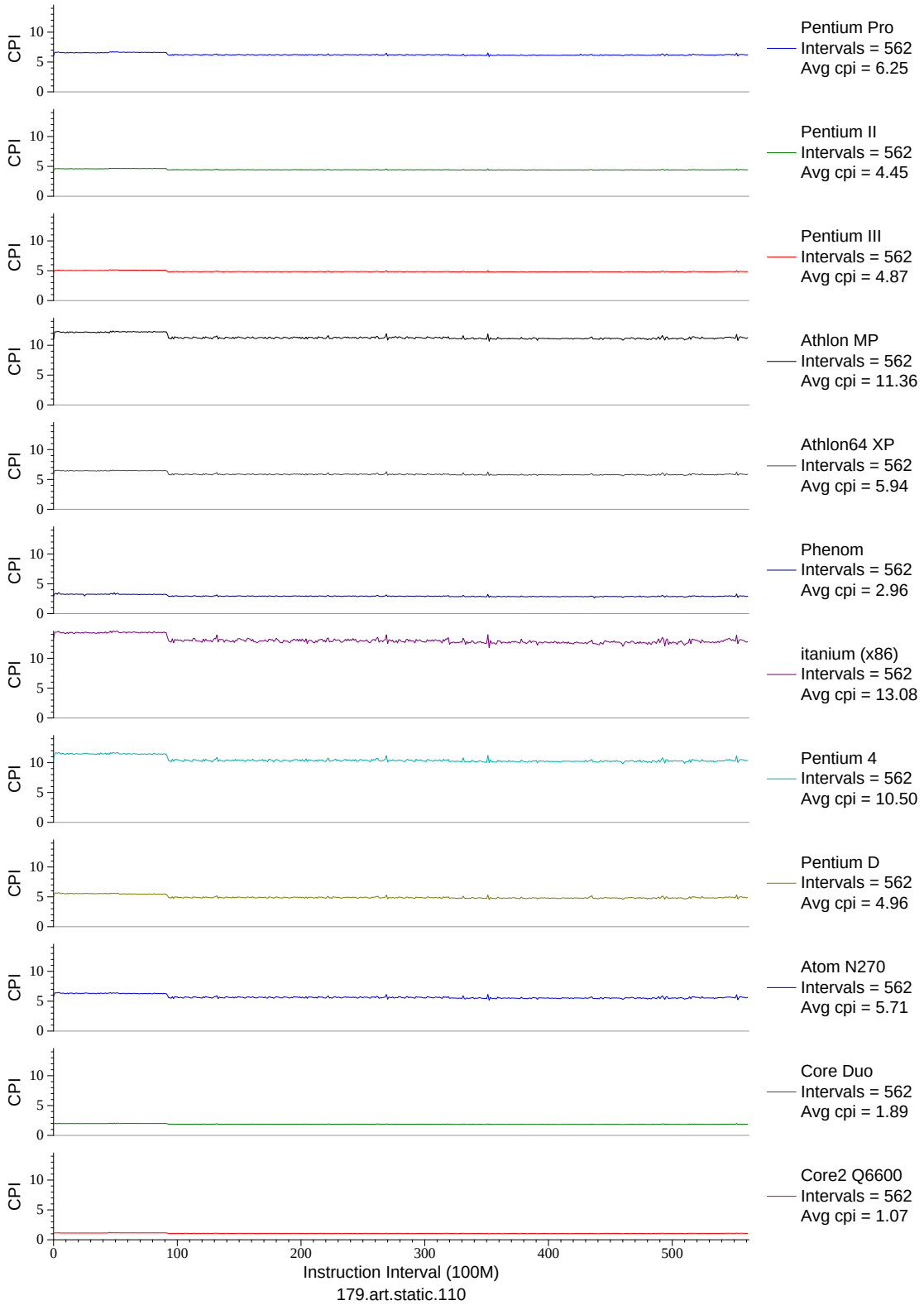

Figure E.19: CPI phase plot for `art.110` (FP, C, Neural Networks)

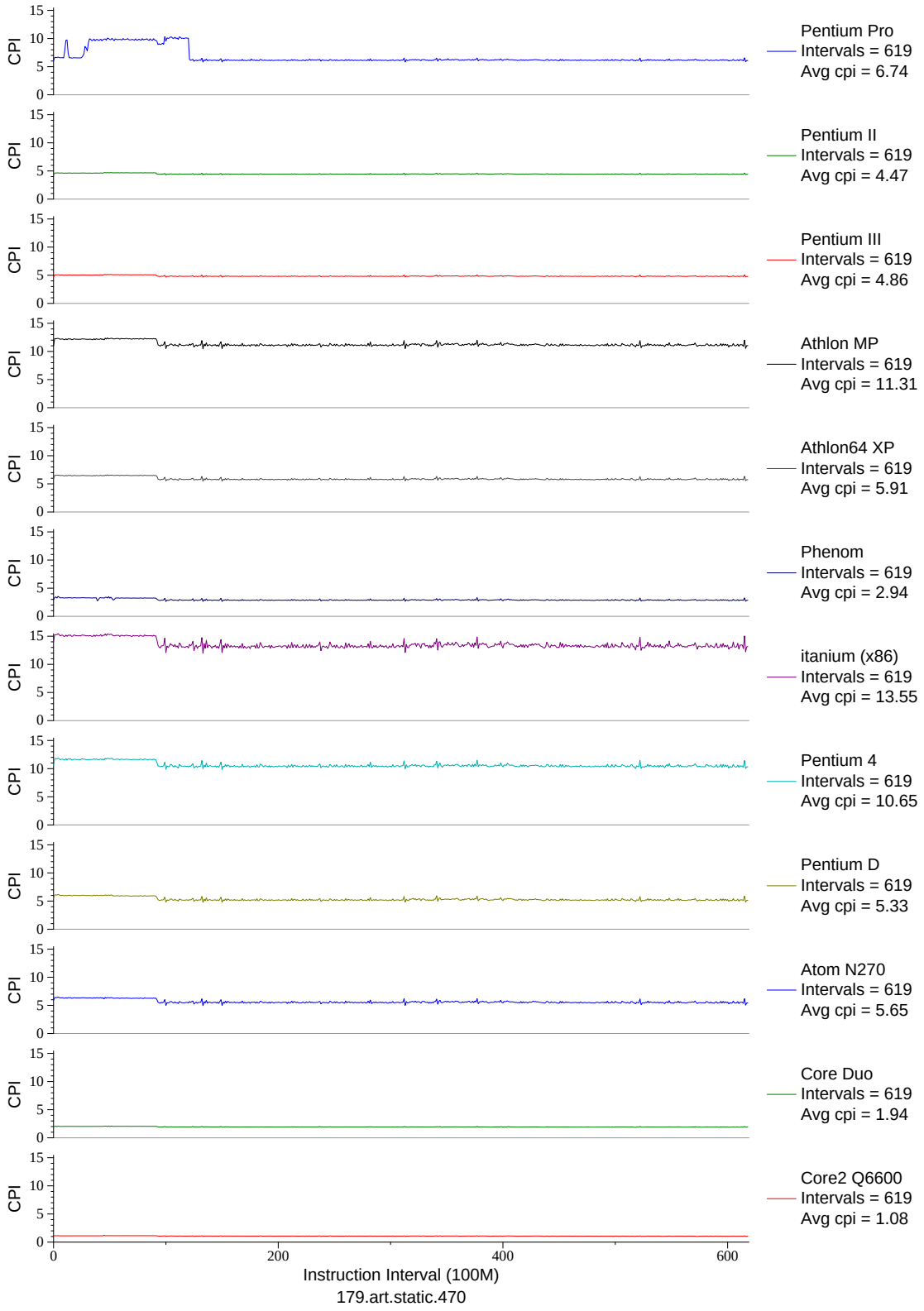

Figure E.20: CPI phase plot for `art.470` (FP, C, Neural Networks)

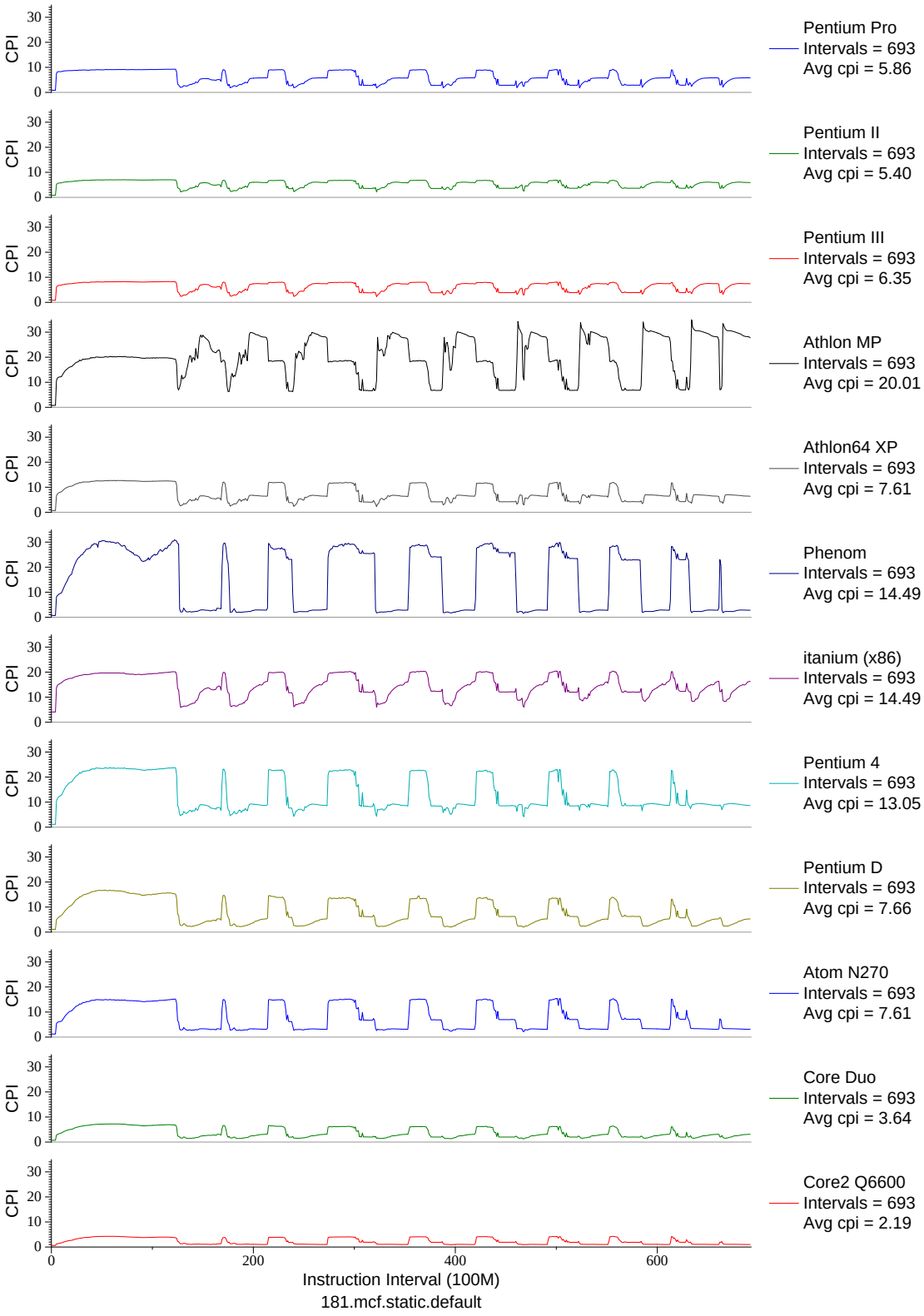

Figure E.21: CPI phase plot for `mcf` (INT, C, Combinatorial Opt)

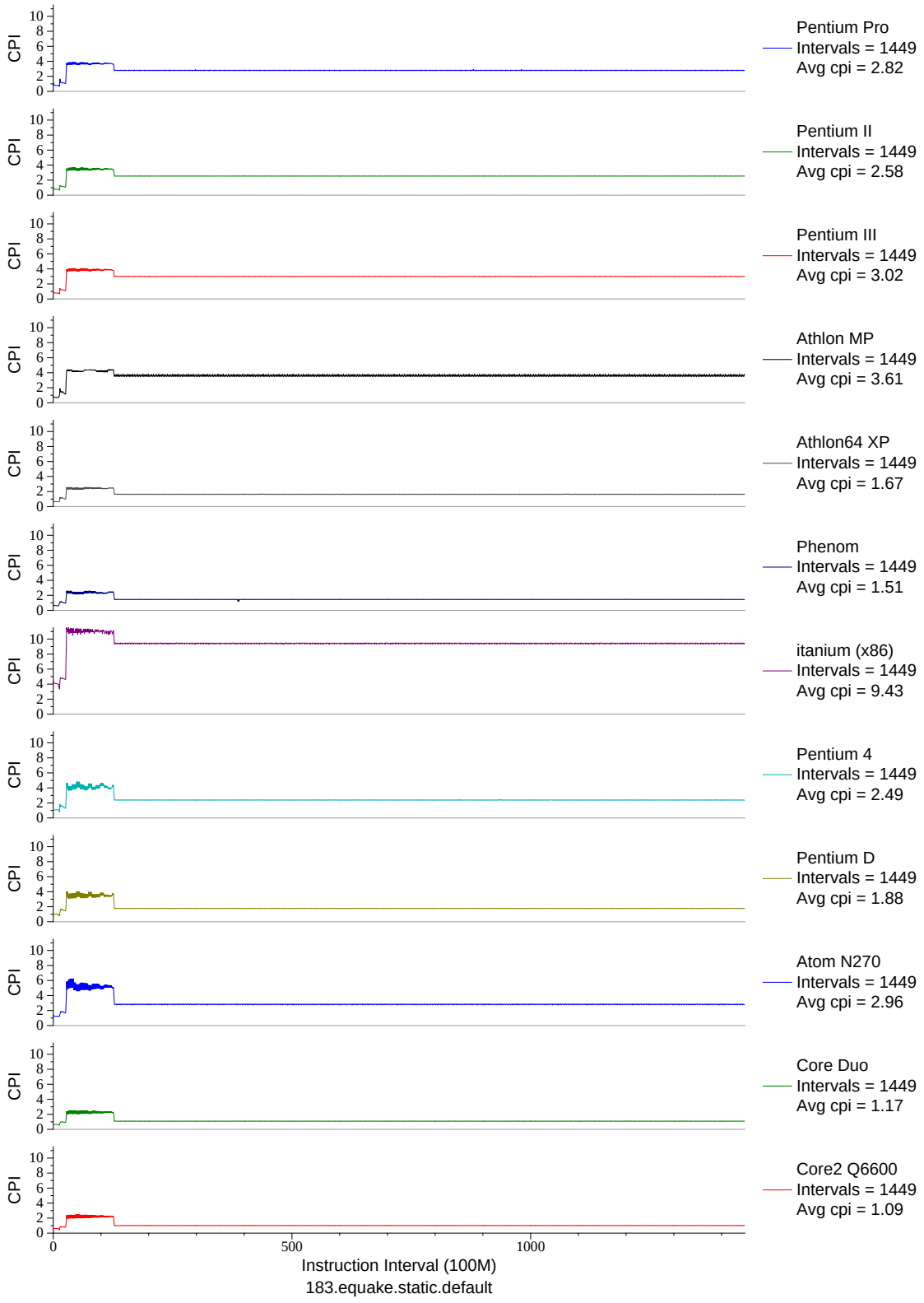

Figure E.22: CPI phase plot for `equake` (FP, C, Seismic Propogation)

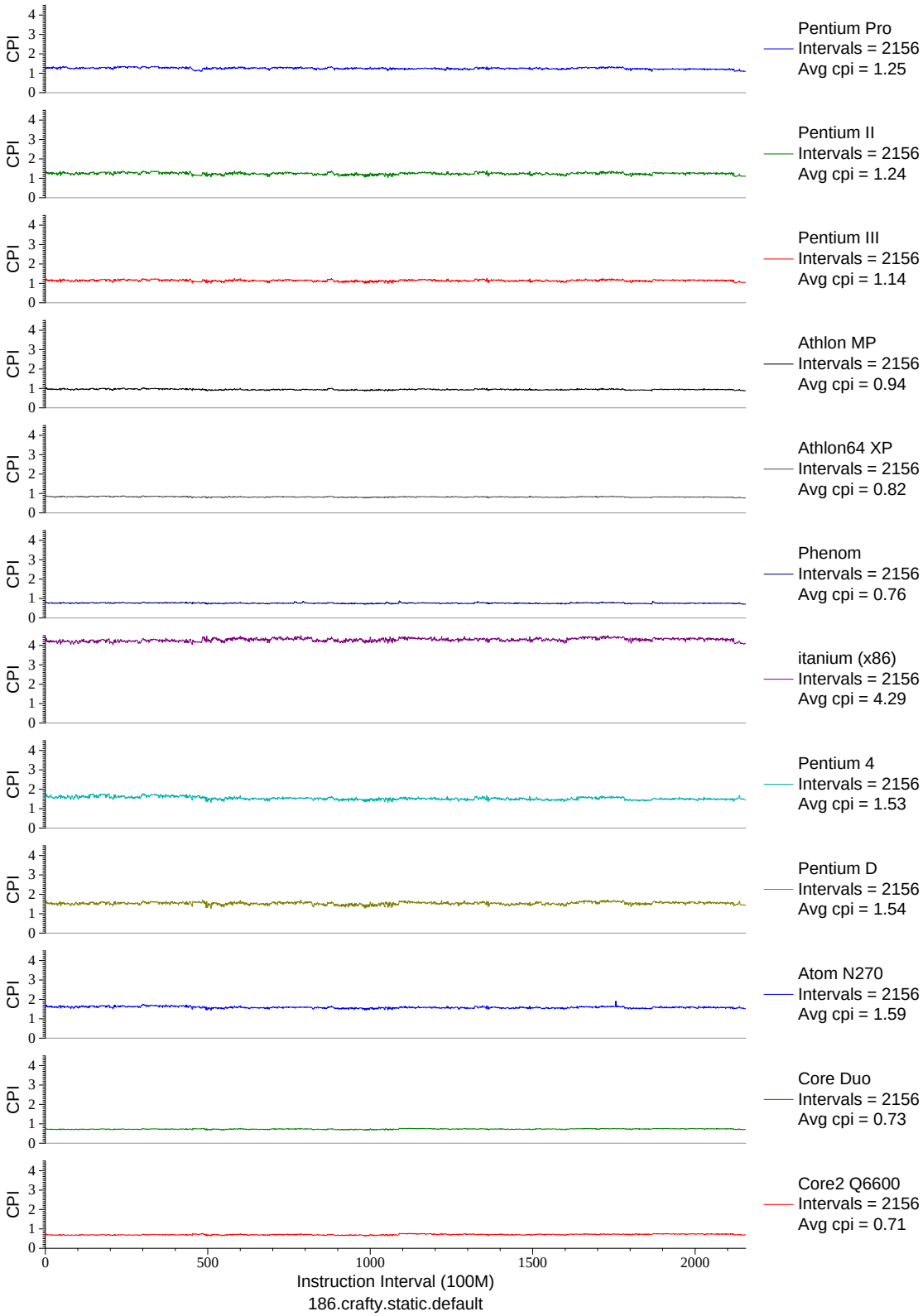

Figure E.23: CPI phase plot for `crafty` (INT, C, Chess)

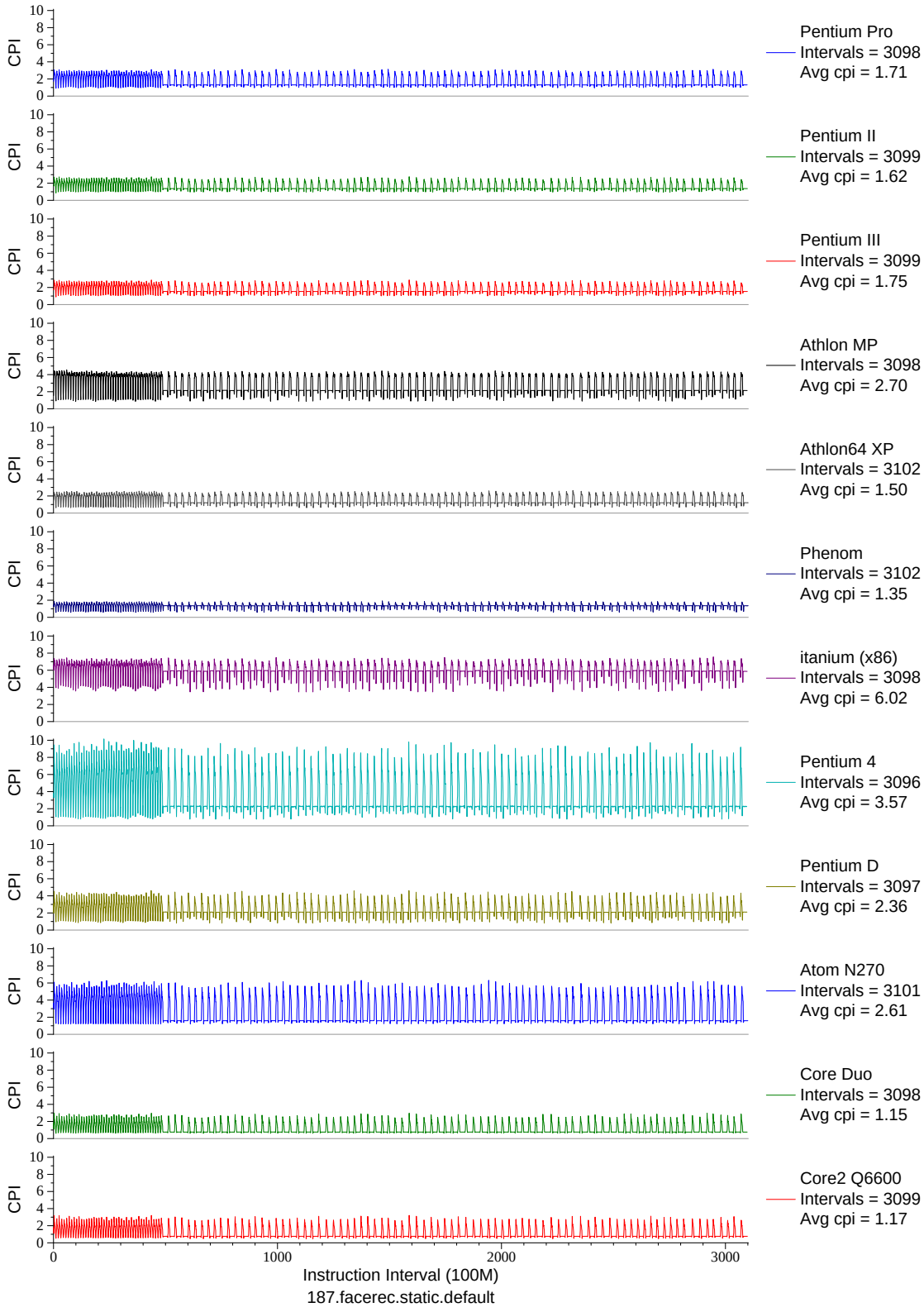

Figure E.24: CPI phase plot for `facerec` (FP, F90, Facial Recognition)

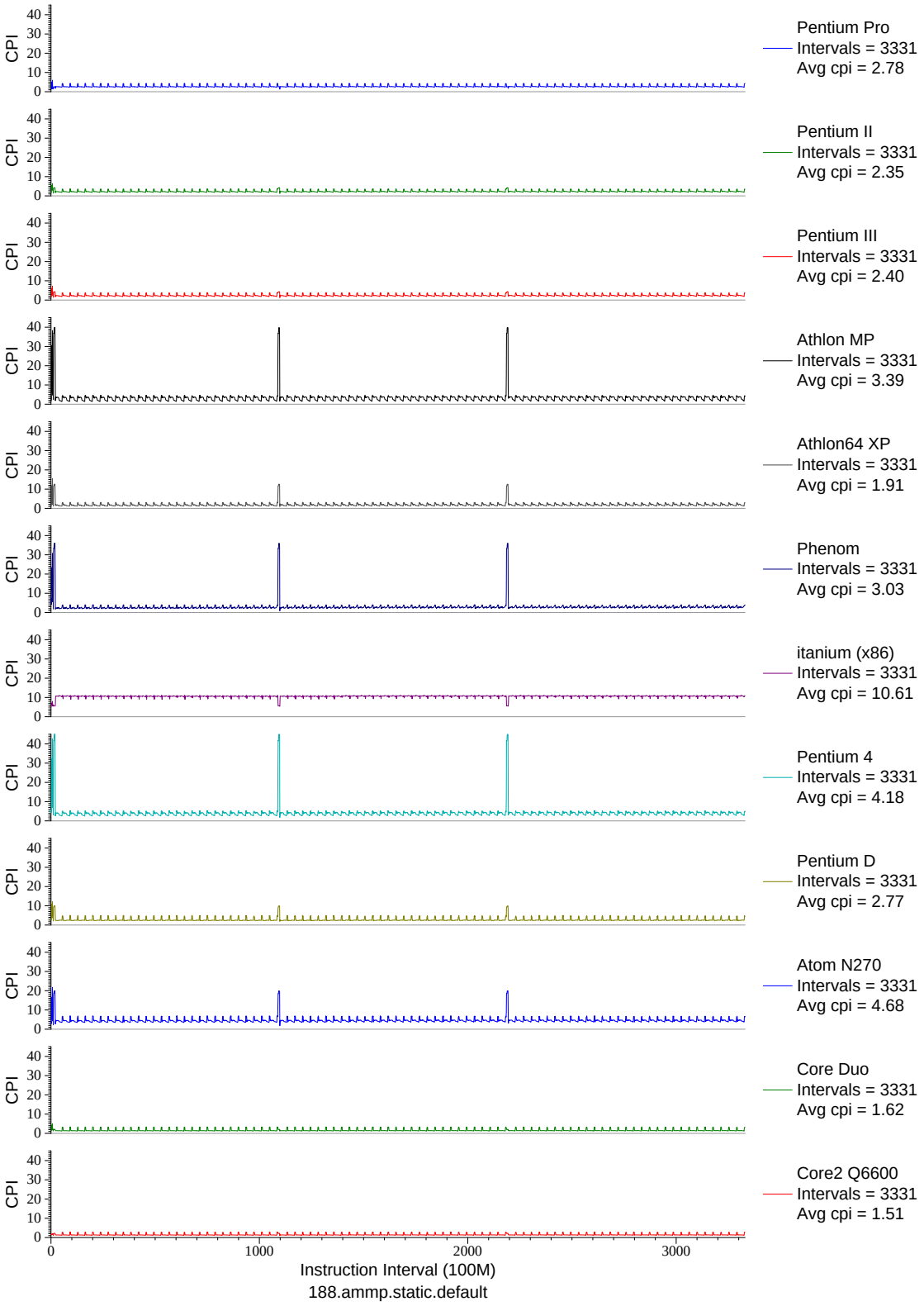

Figure E.25: CPI phase plot for `ammp` (FP, C, Chemistry)

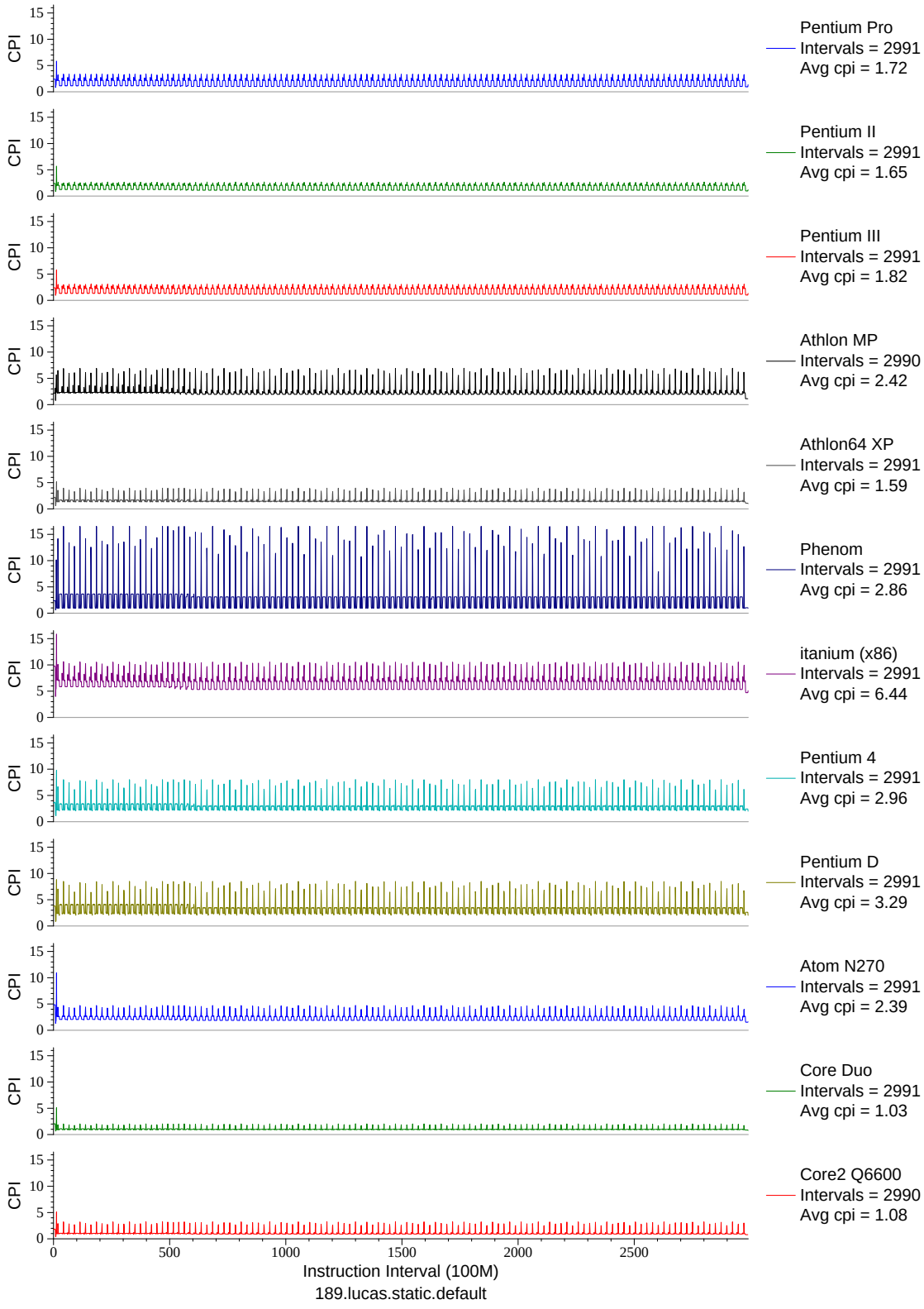

Figure E.26: CPI phase plot for `lucas` (FP, F90, Number Theory)

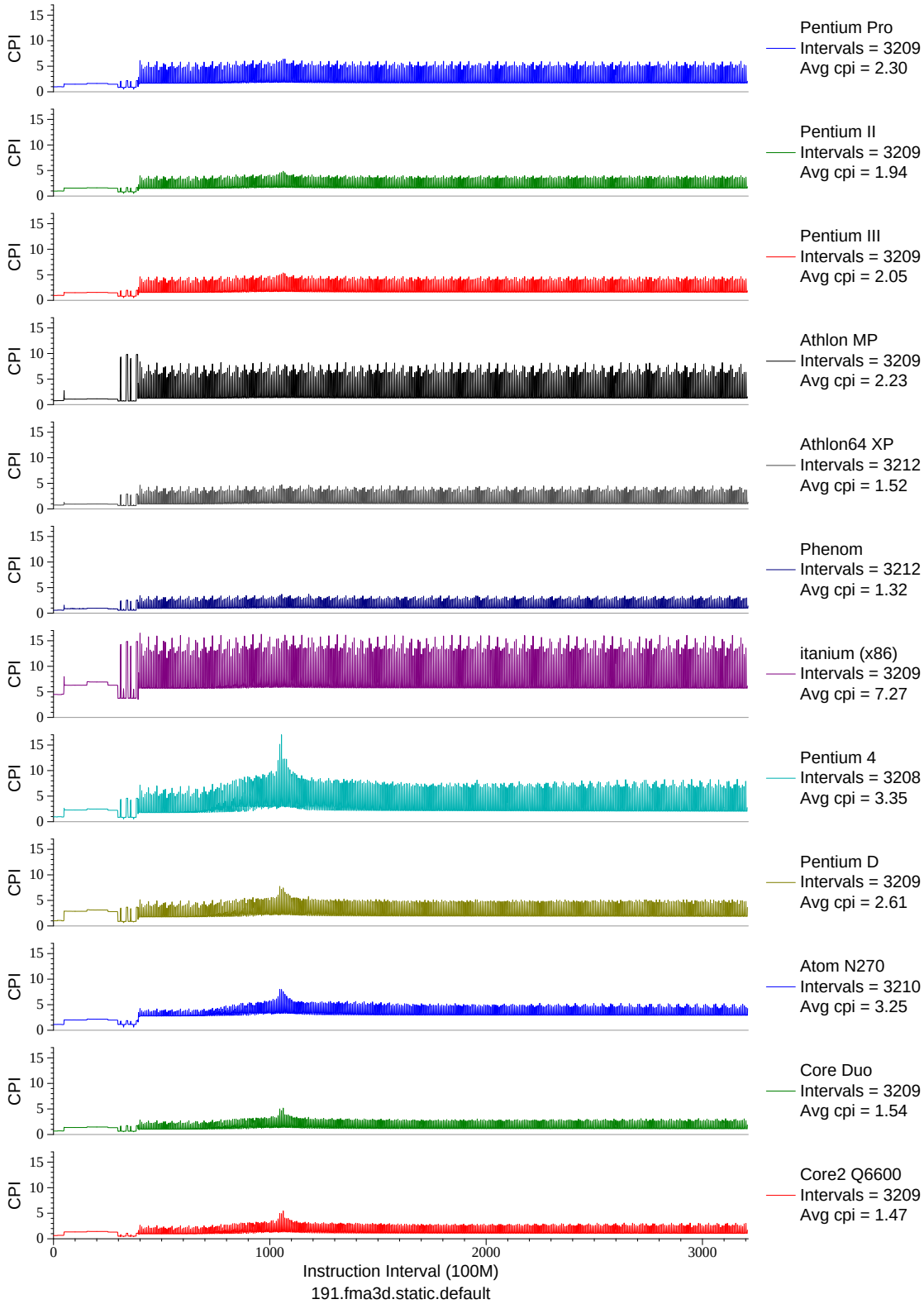

Figure E.27: CPI phase plot for `fma3d` (FP, F90, Crash Simulation)

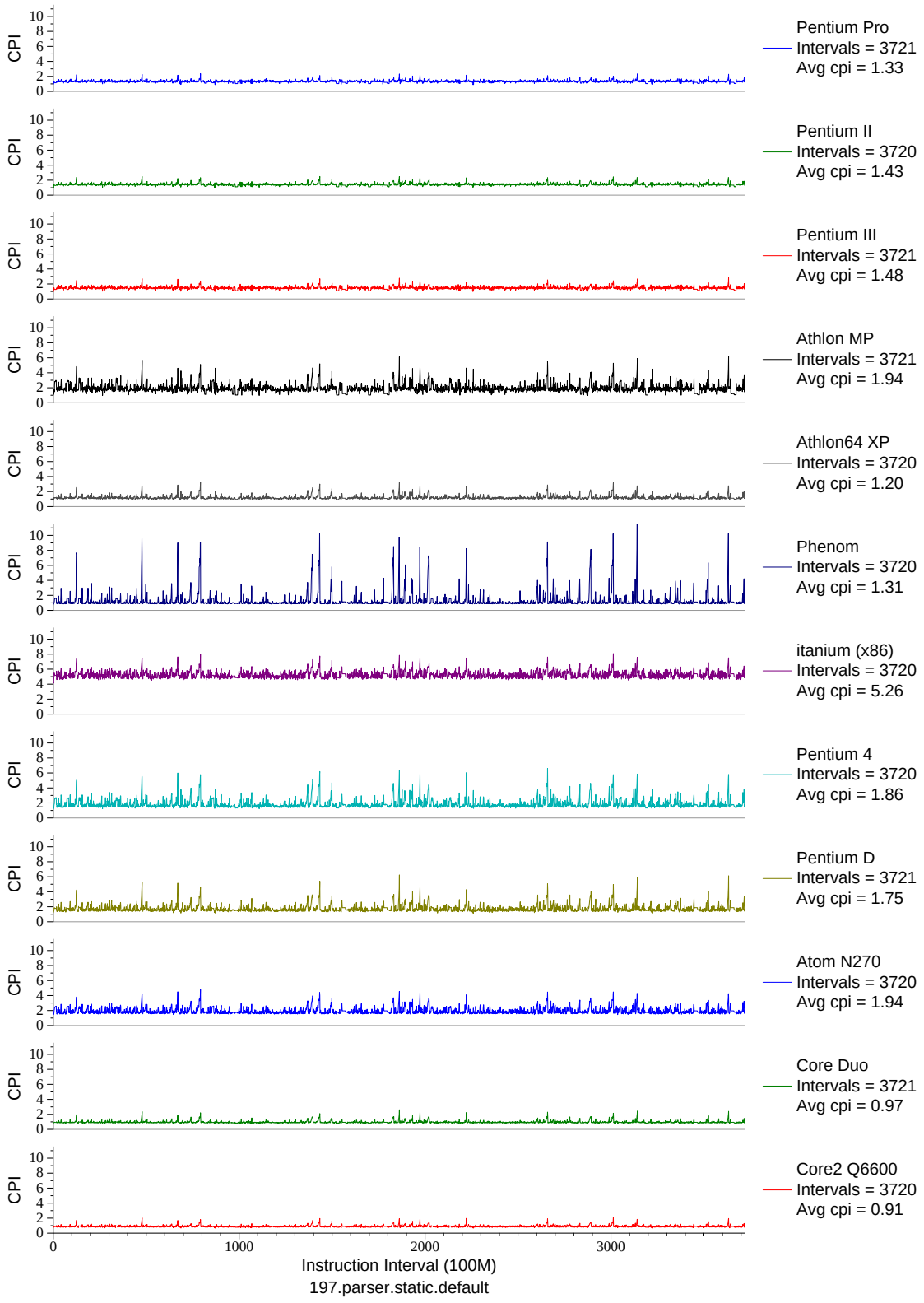

Figure E.28: CPI phase plot for `parser` (INT, C, Word Processing)

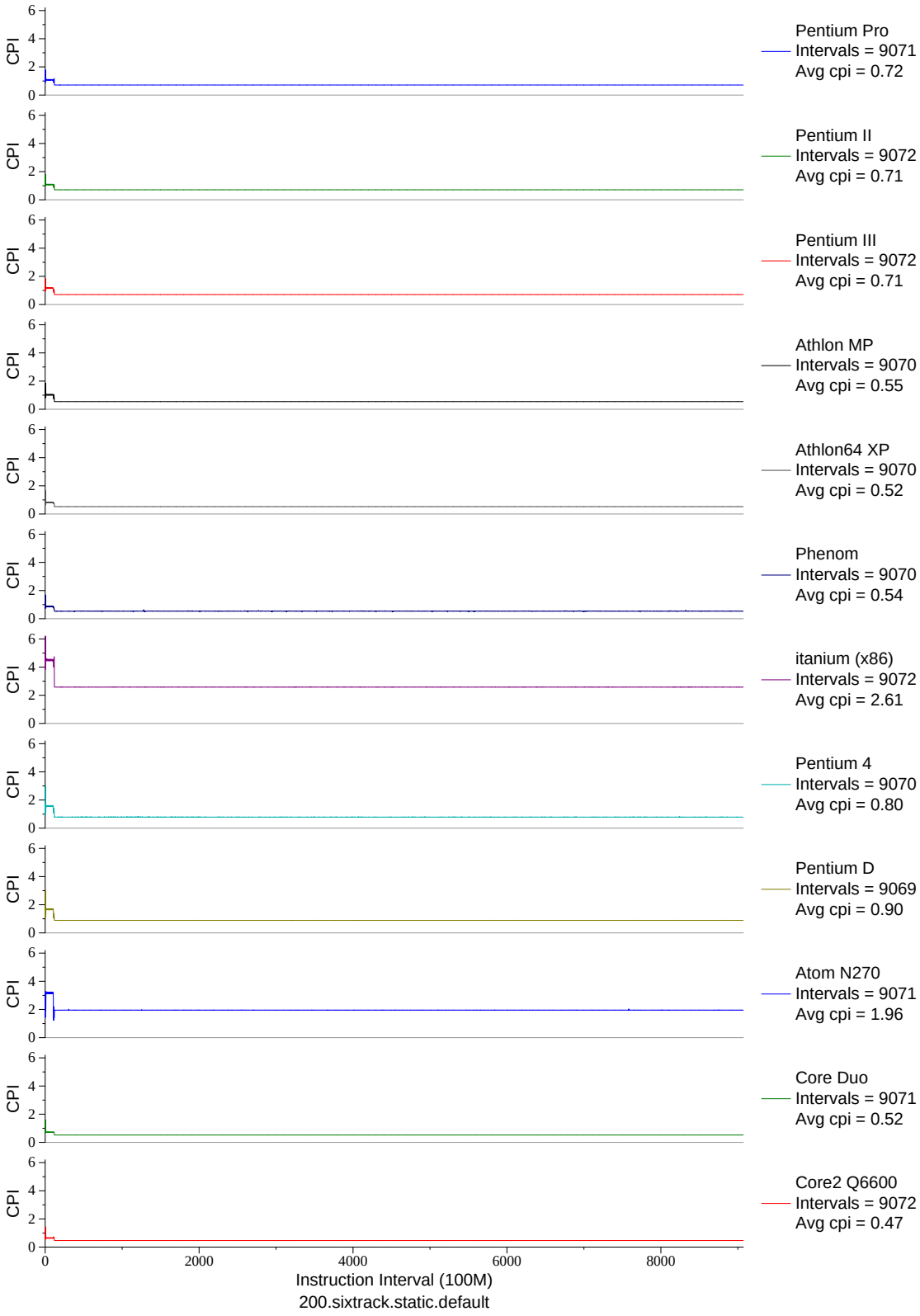

Figure E.29: CPI phase plot for `sixtrack` (FP, F77, Nuclear Physics)

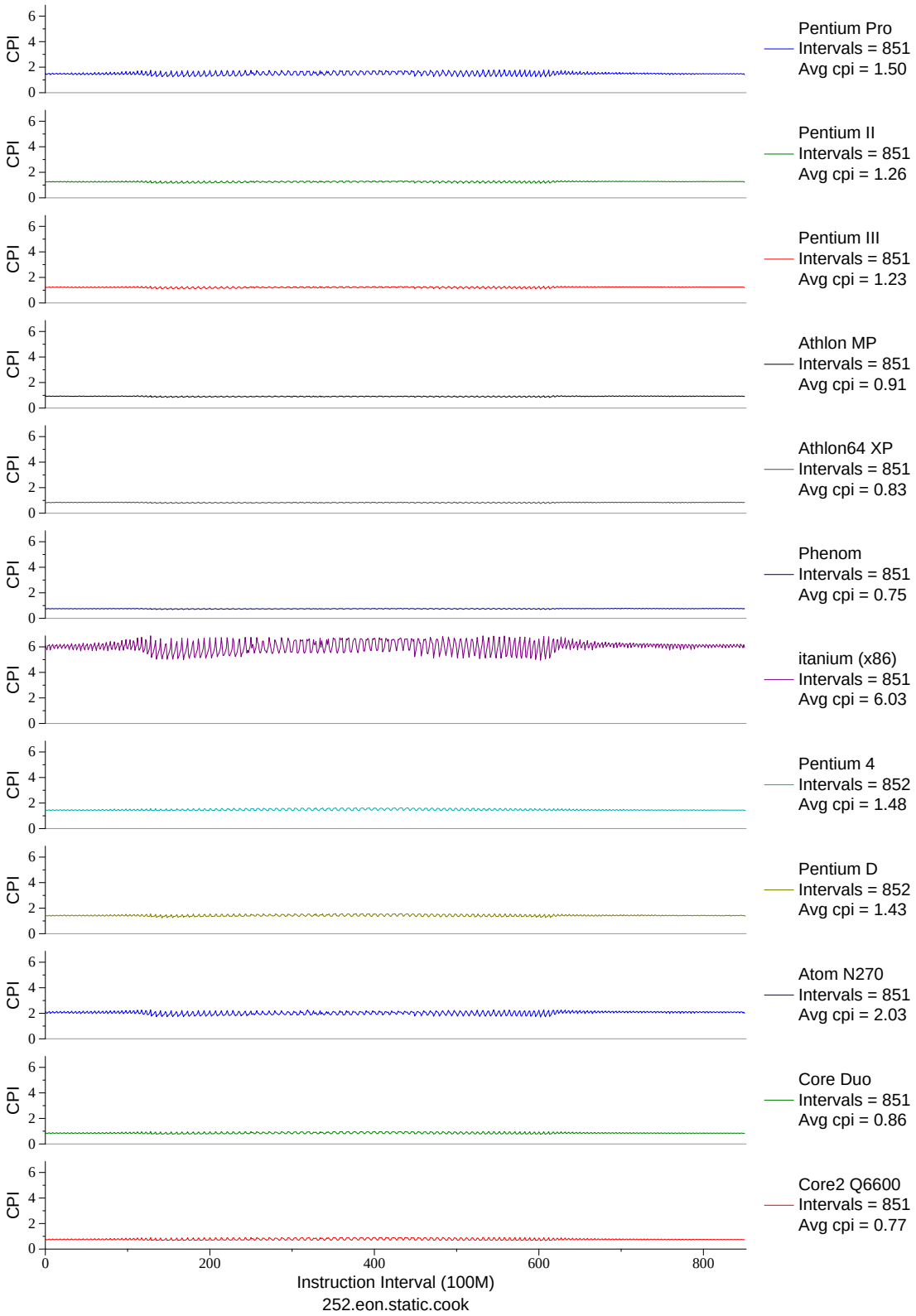

Figure E.30: CPI phase plot for `eon.cook` (INT, C++, Computer Graphics)

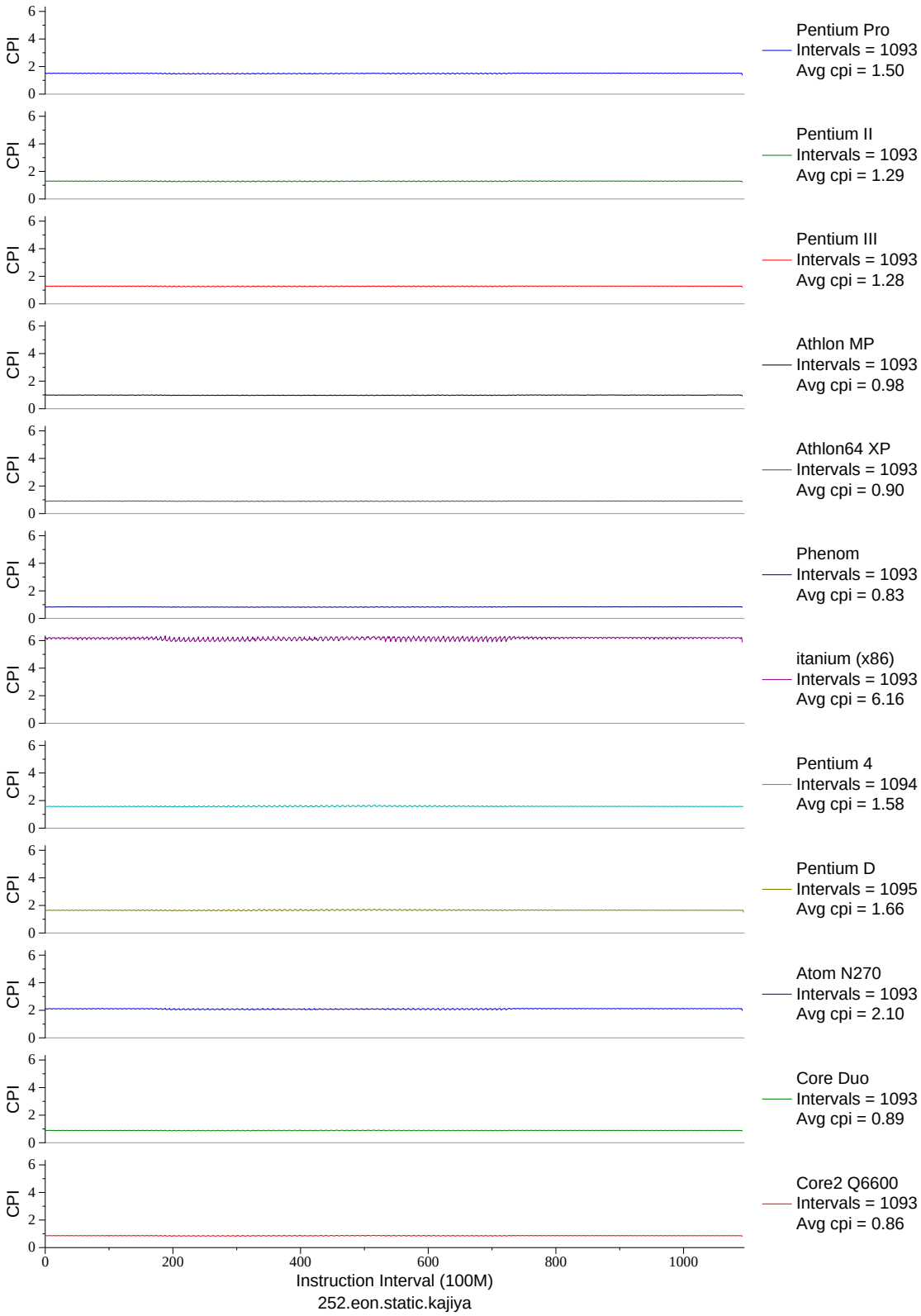

Figure E.31: CPI phase plot for `eon.kaj` (INT, C++, Computer Graphics)

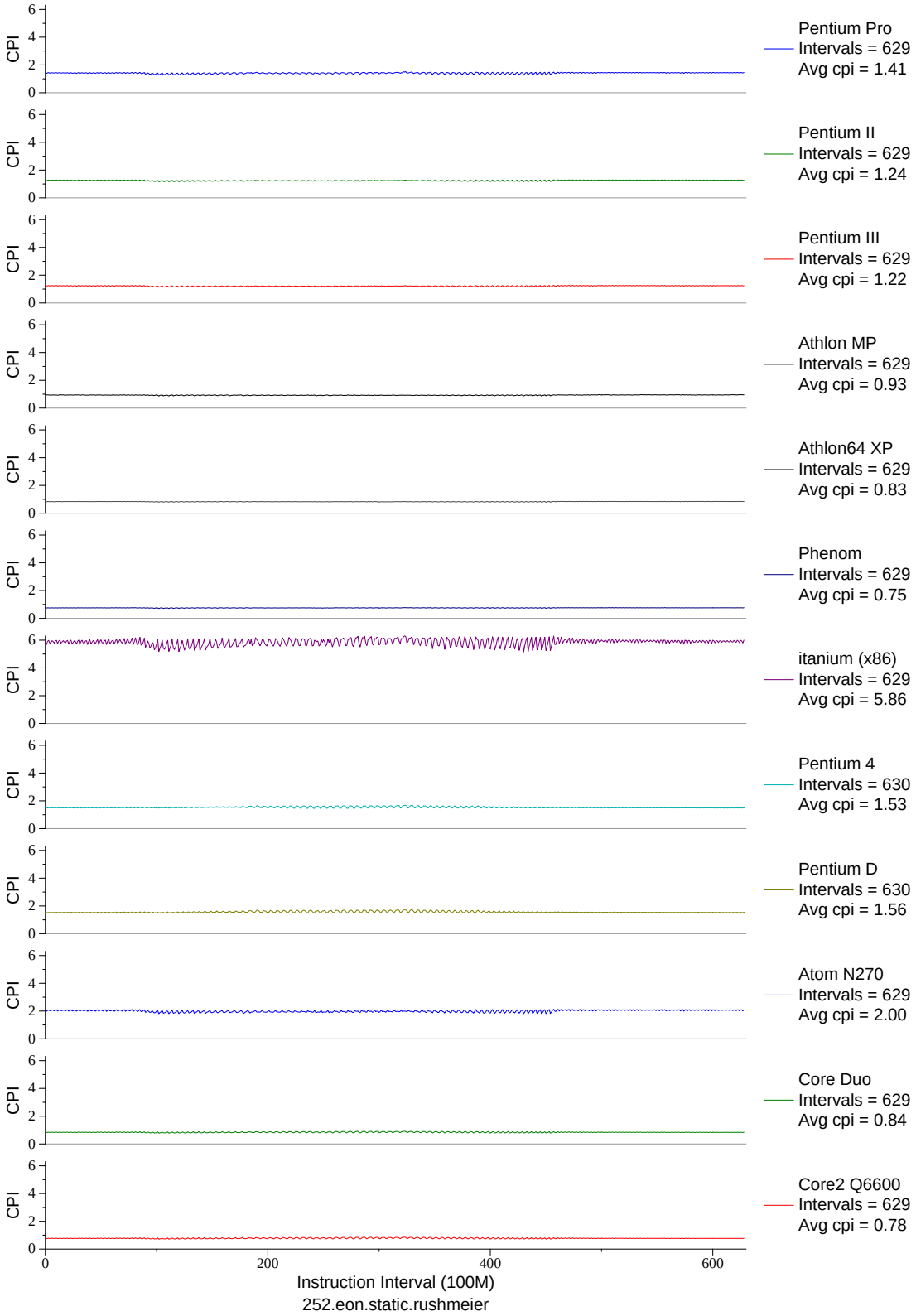

Figure E.32: CPI phase plot for `eon.rush` (INT, C++, Computer Graphics)

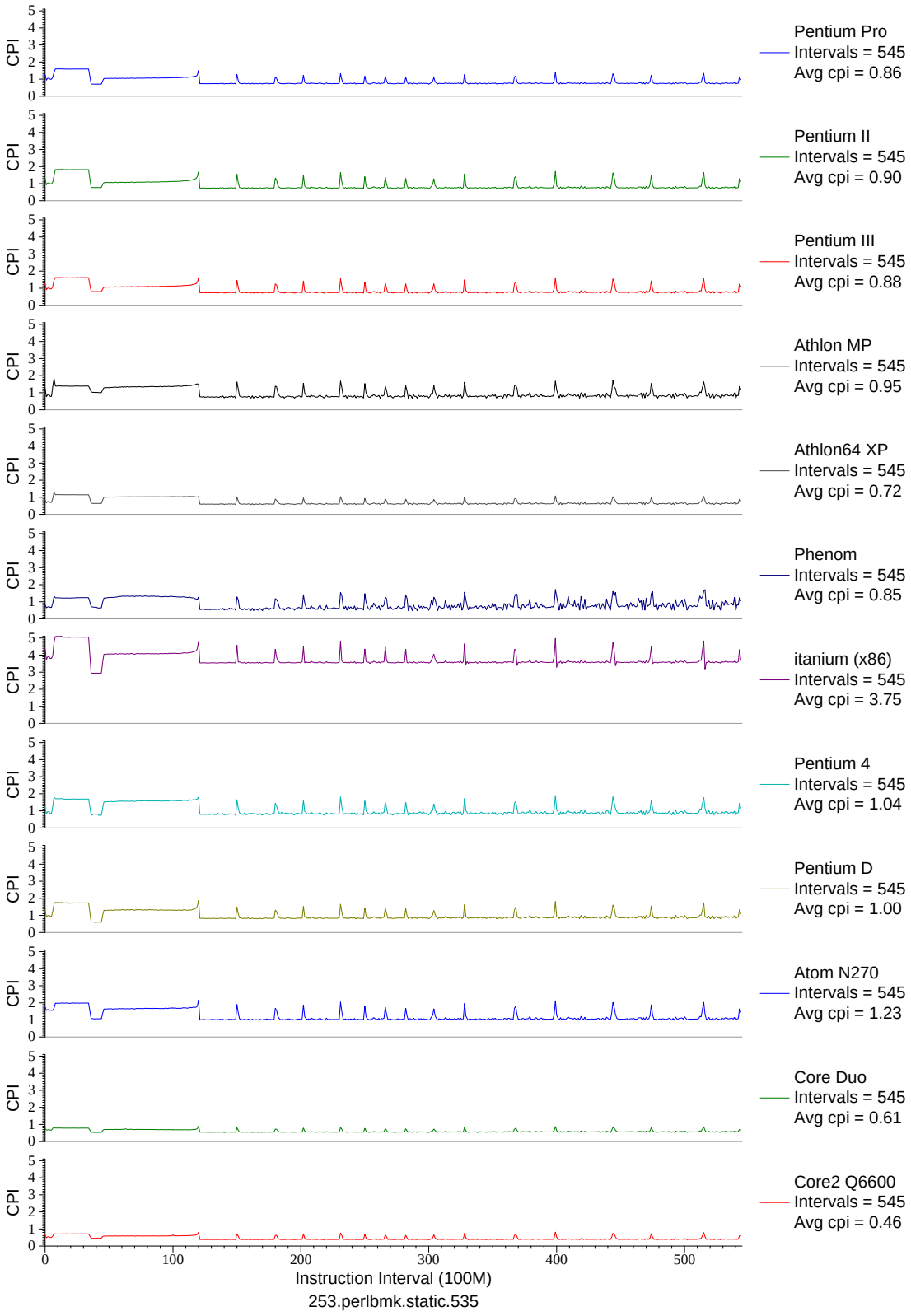

Figure E.33: CPI phase plot for `perlbmk.535` (INT, C, Scripting Language)

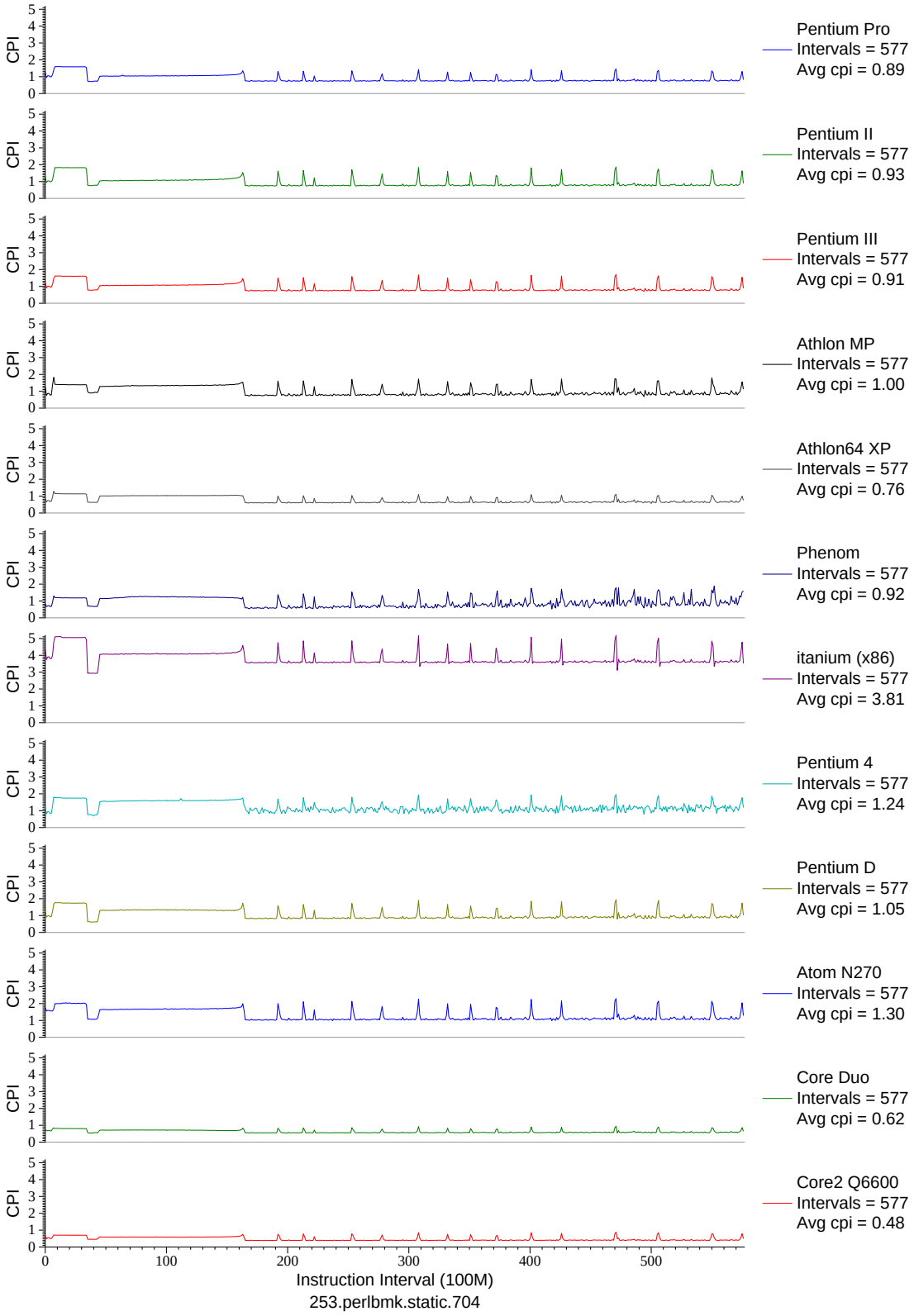

Figure E.34: CPI phase plot for `perlbmk.704` (INT, C, Scripting Language)

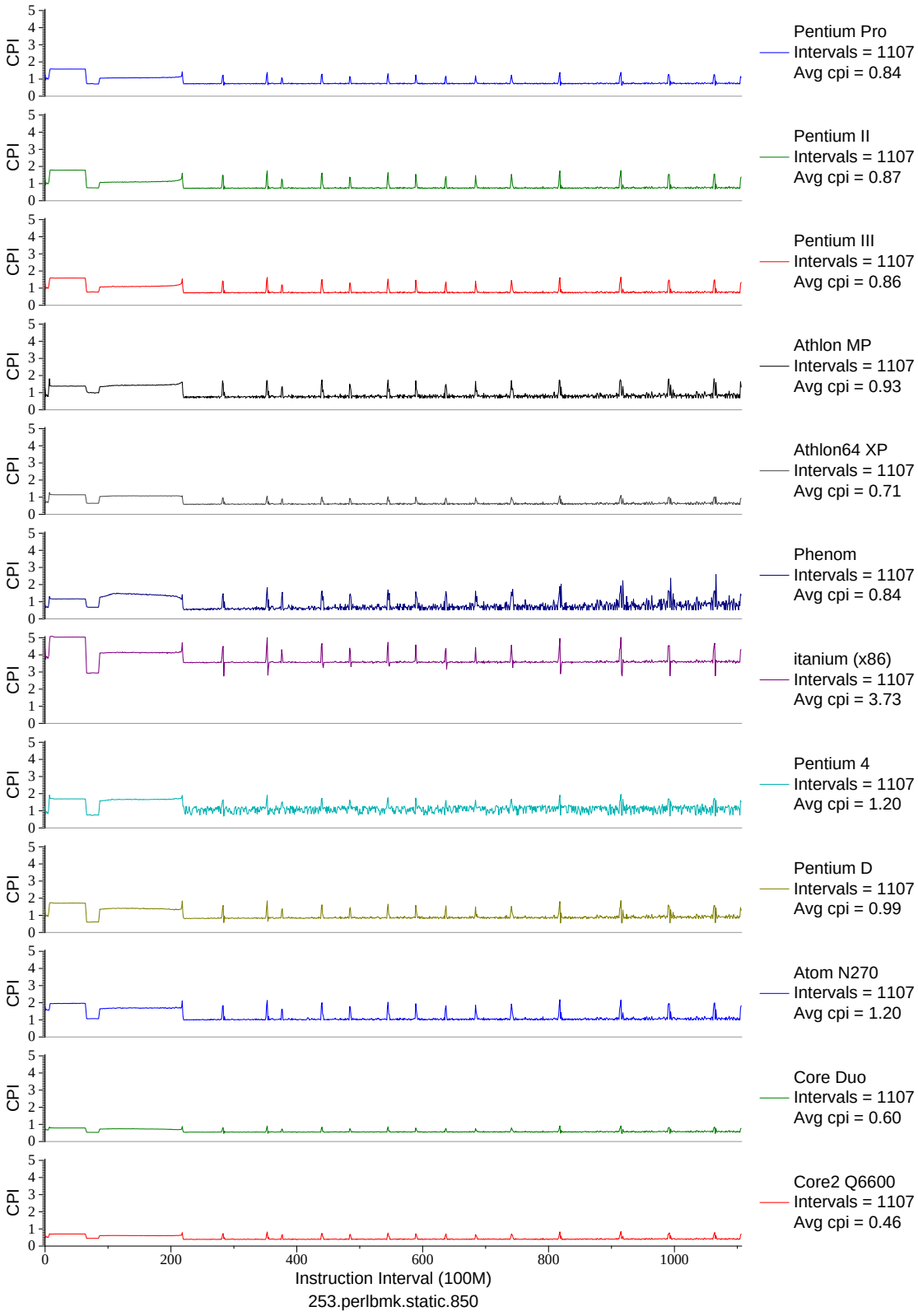

Figure E.35: CPI phase plot for `perlbmk.850` (INT, C, Scripting Language)

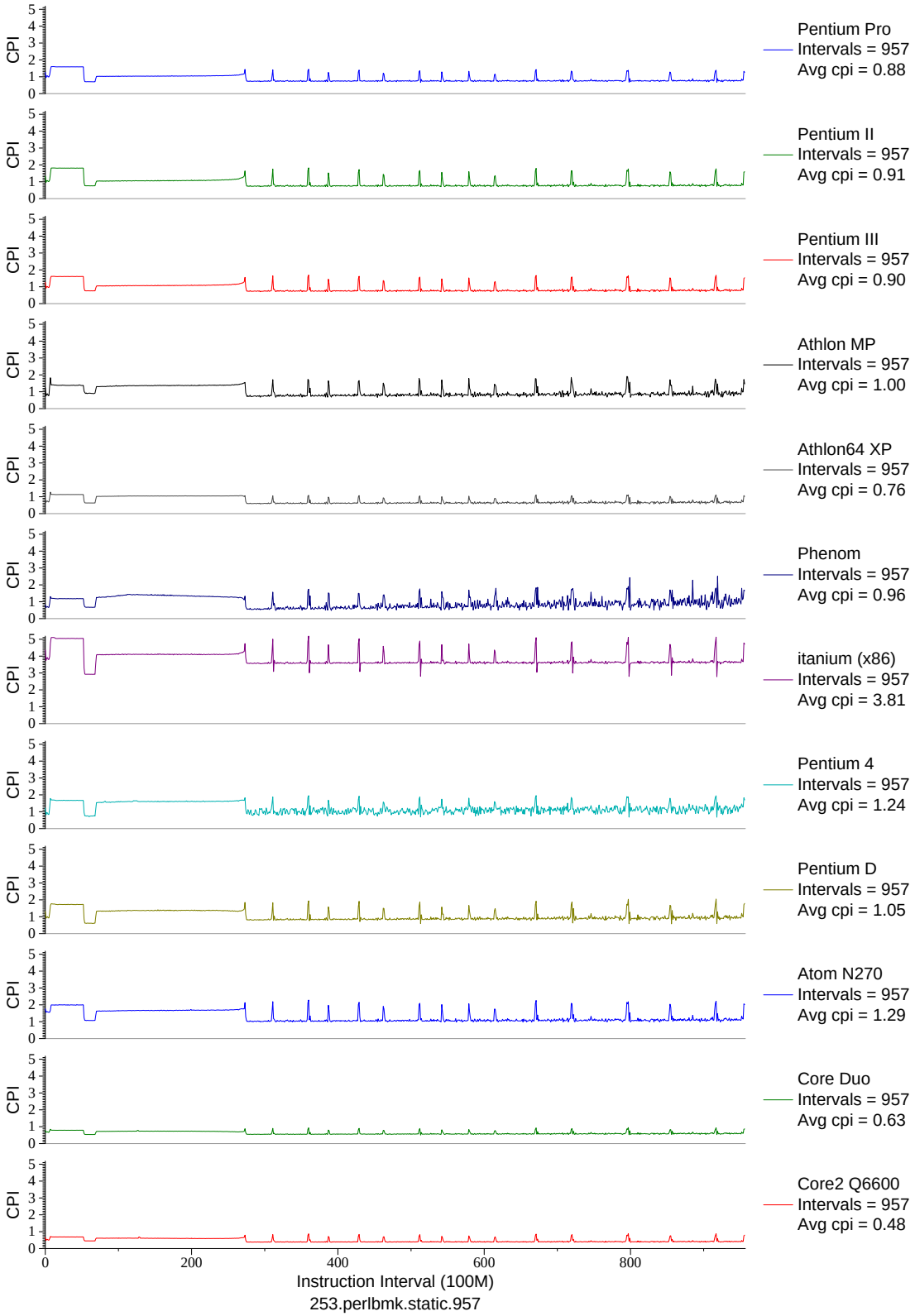

Figure E.36: CPI phase plot for `perlbmk.957` (INT, C, Scripting Language)

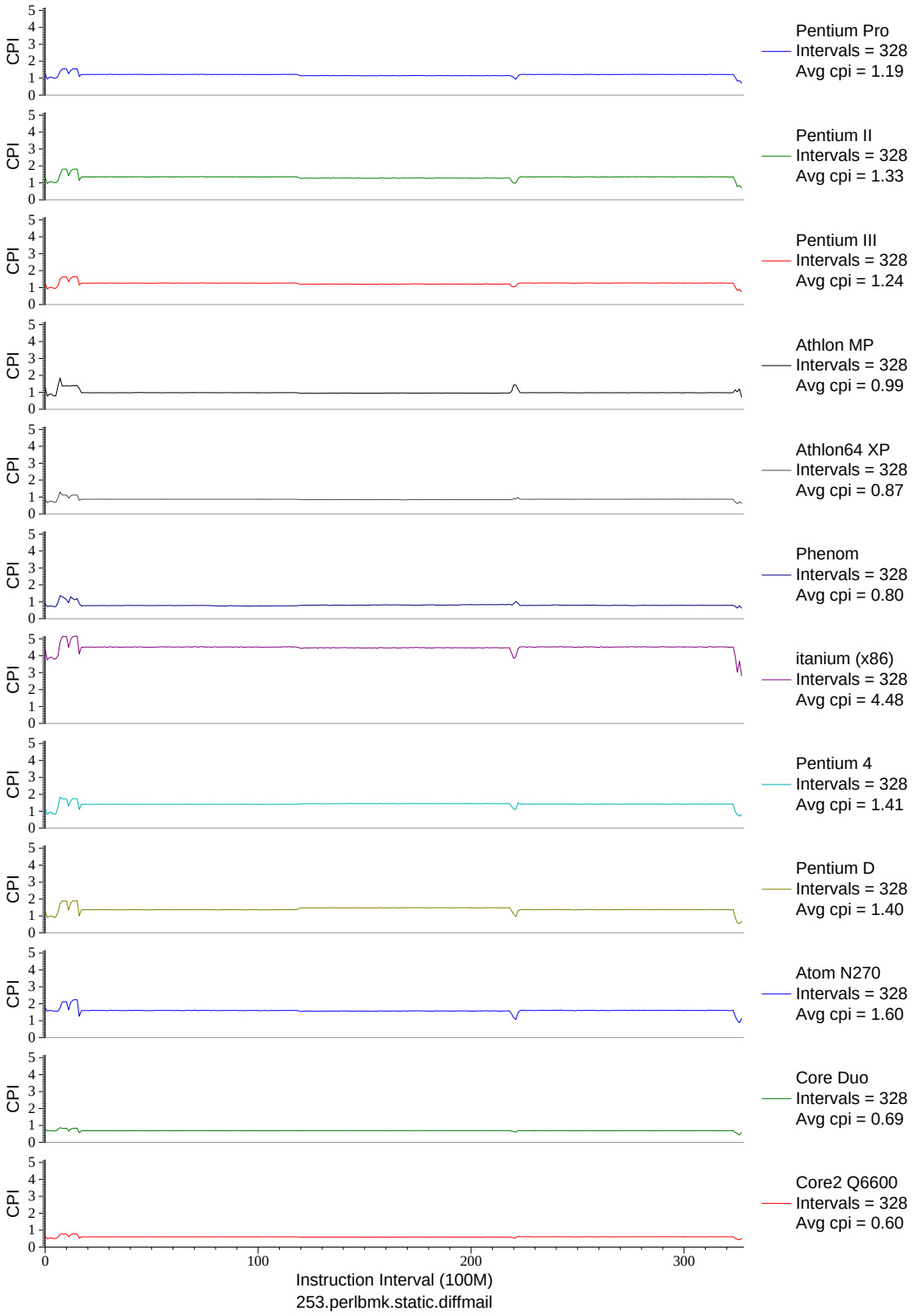

Figure E.37: CPI phase plot for `perlbmk.diff` (INT, C, Scripting Language)

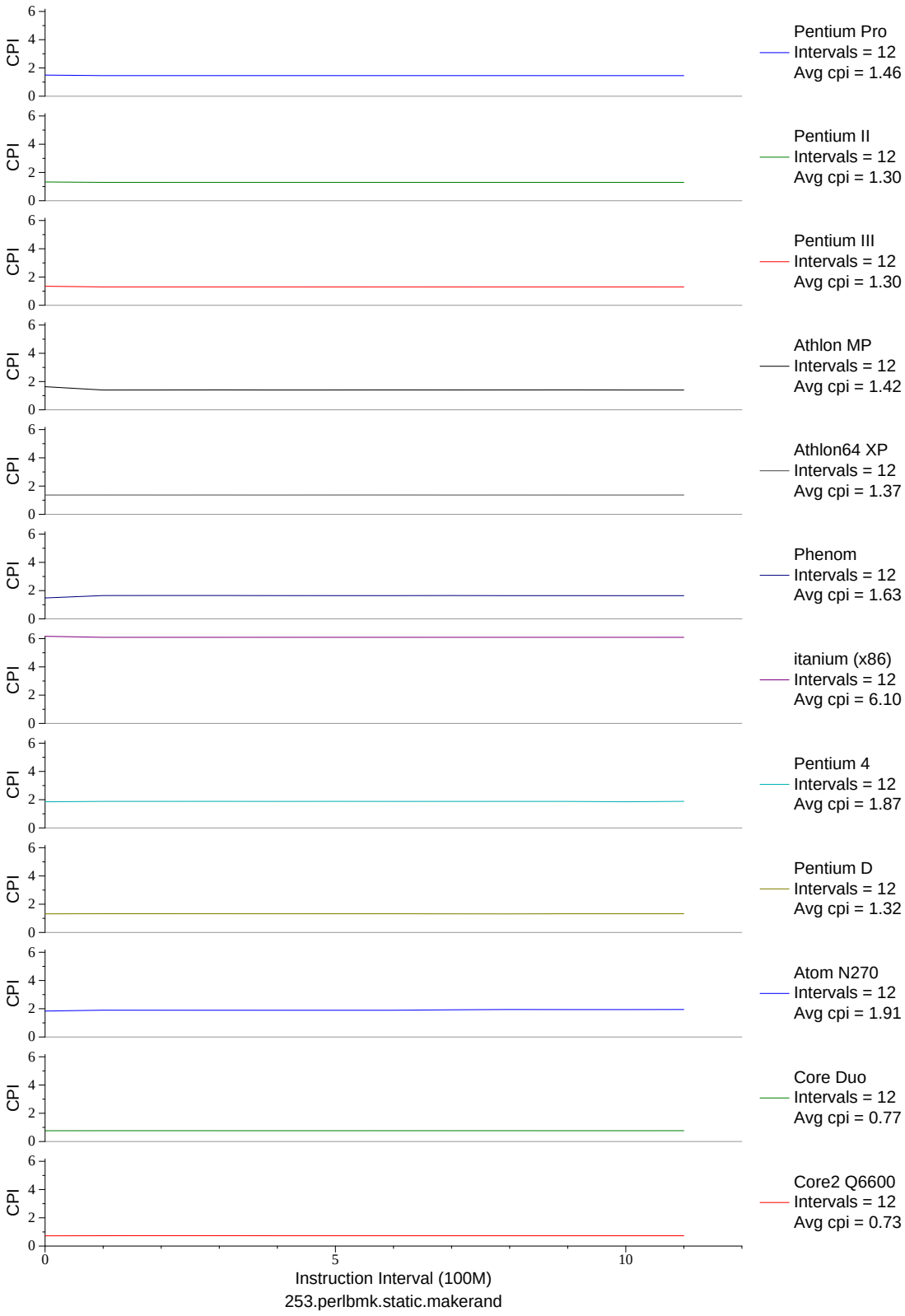

Figure E.38: CPI phase plot for `perlbmk.mkrnd` (INT, C, Scripting Language)

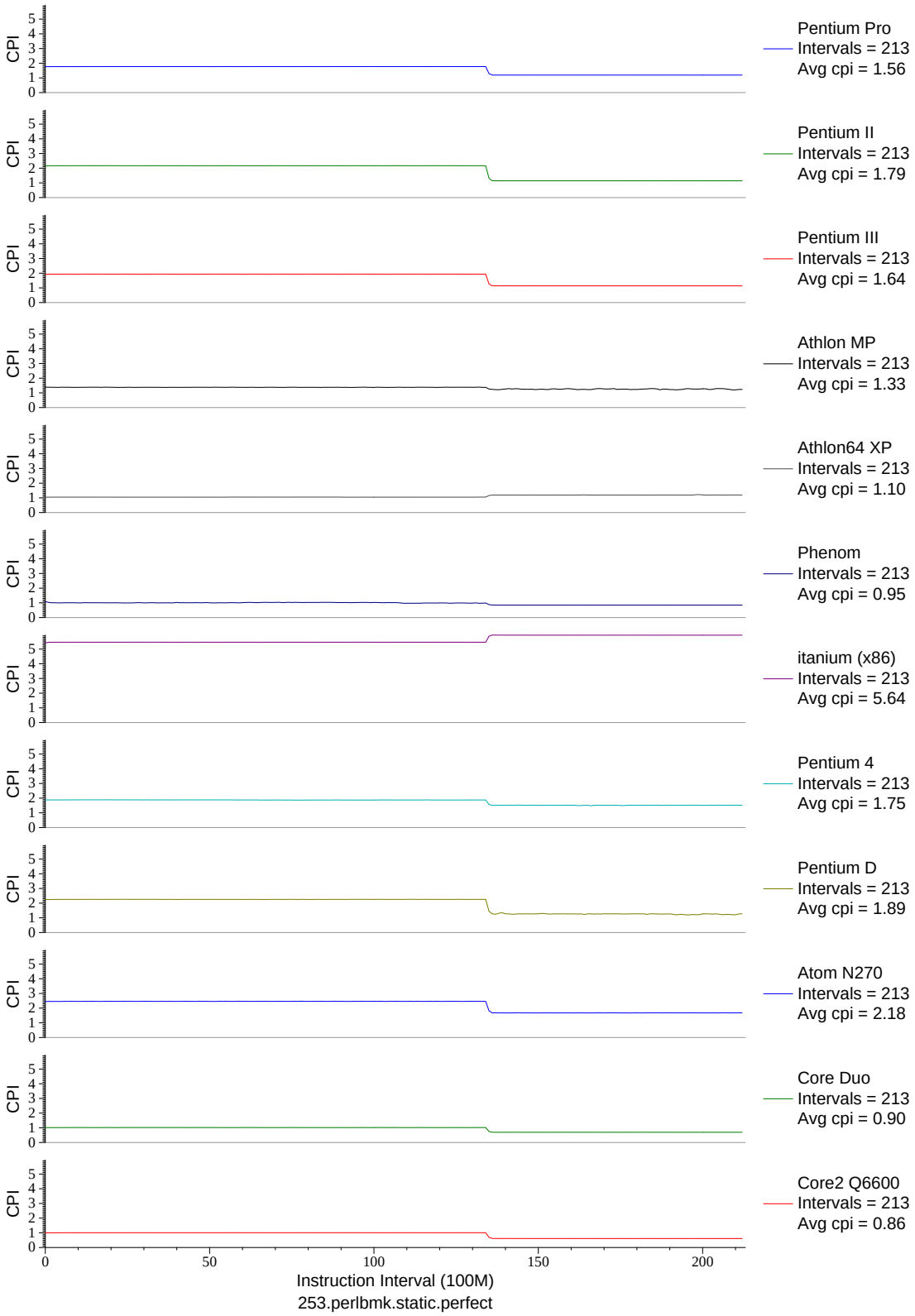

Figure E.39: CPI phase plot for `perlbmk.perf` (INT, C, Scripting Language)

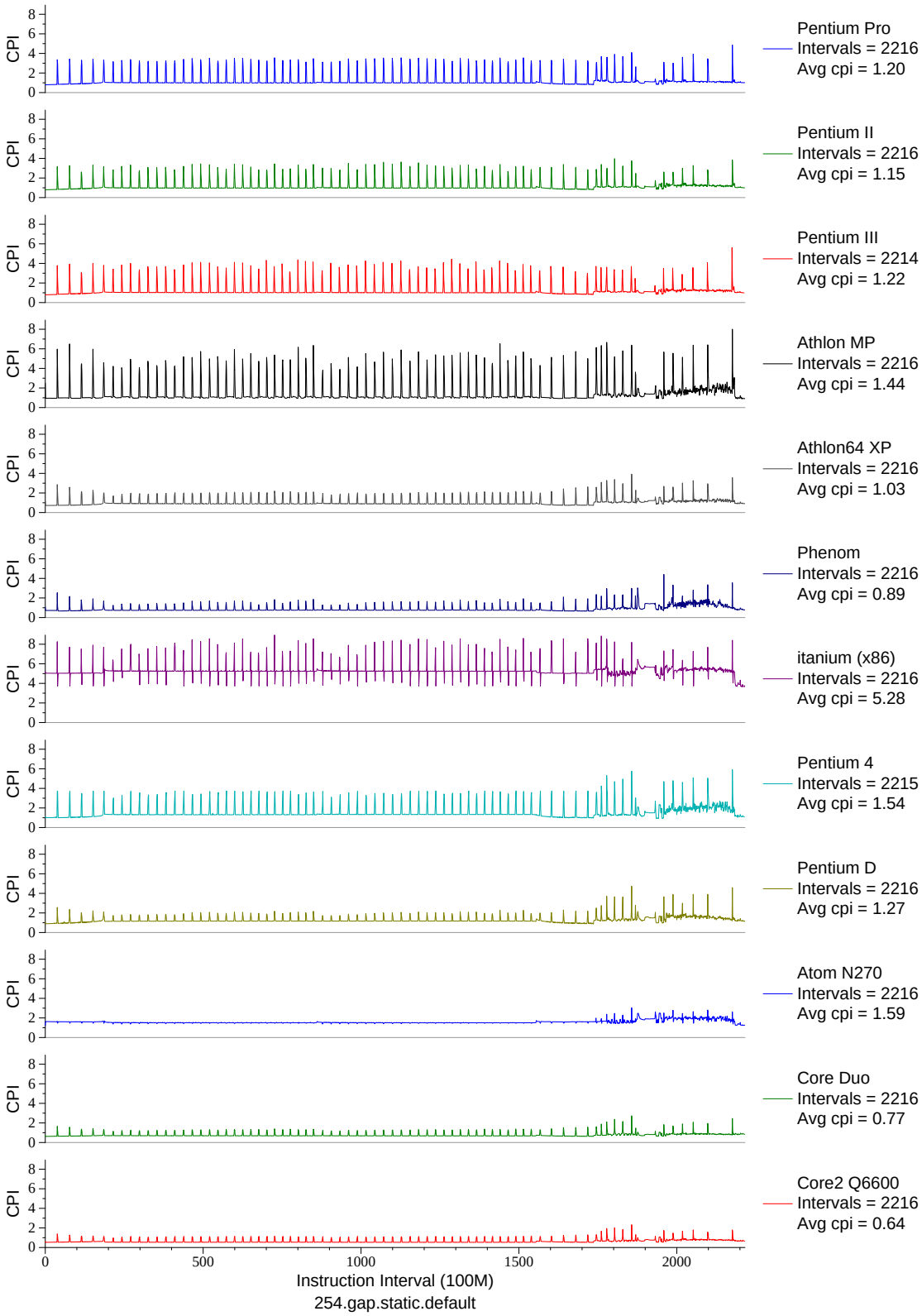

Figure E.40: CPI phase plot for `gap` (INT, C, Group Theory)

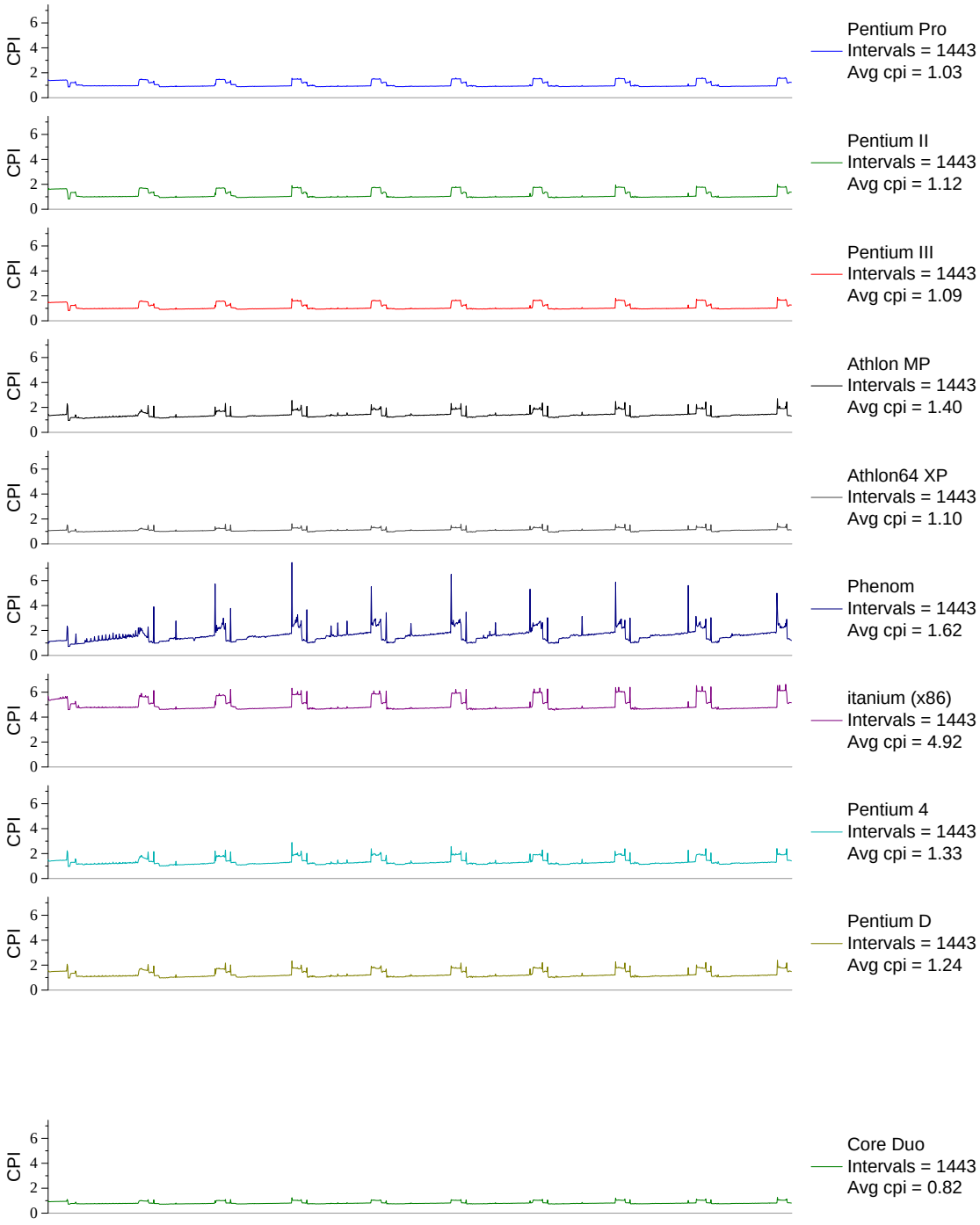

Figure E.41: CPI phase plot for `vortex.1` (INT, C, Database)

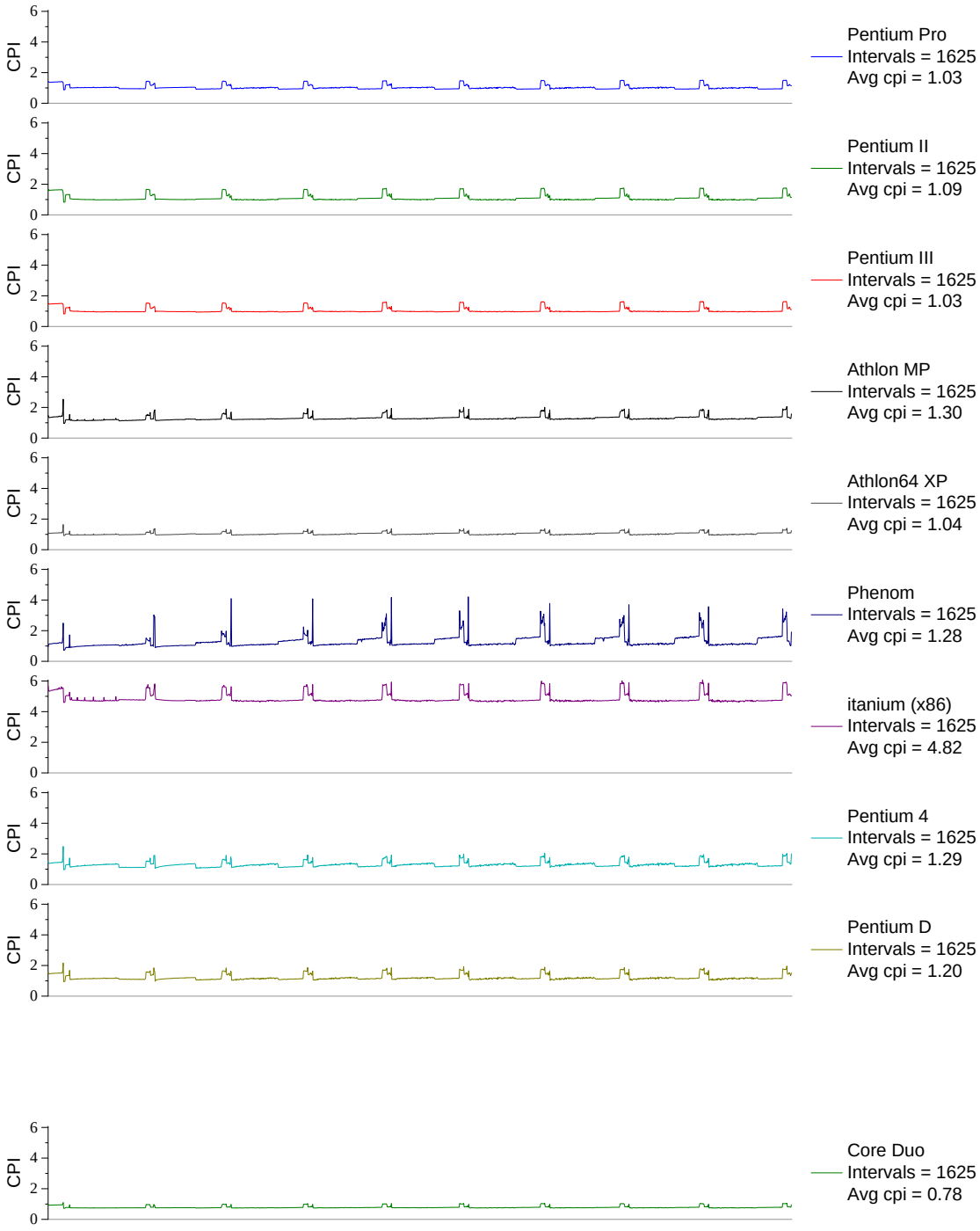

Figure E.42: CPI phase plot for `vortex.2` (INT, C, Database)

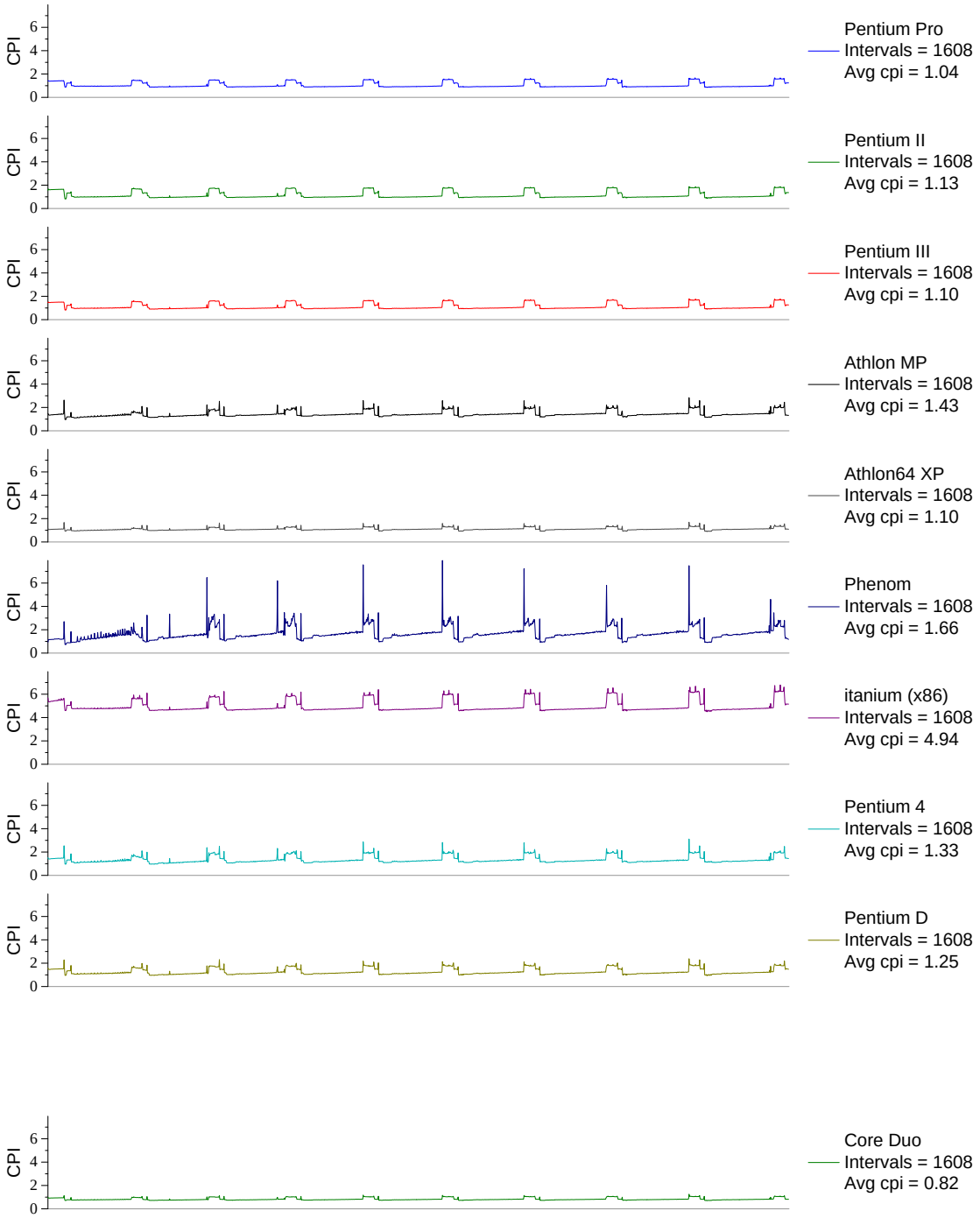

Figure E.43: CPI phase plot for `vortex.3` (INT, C, Database)

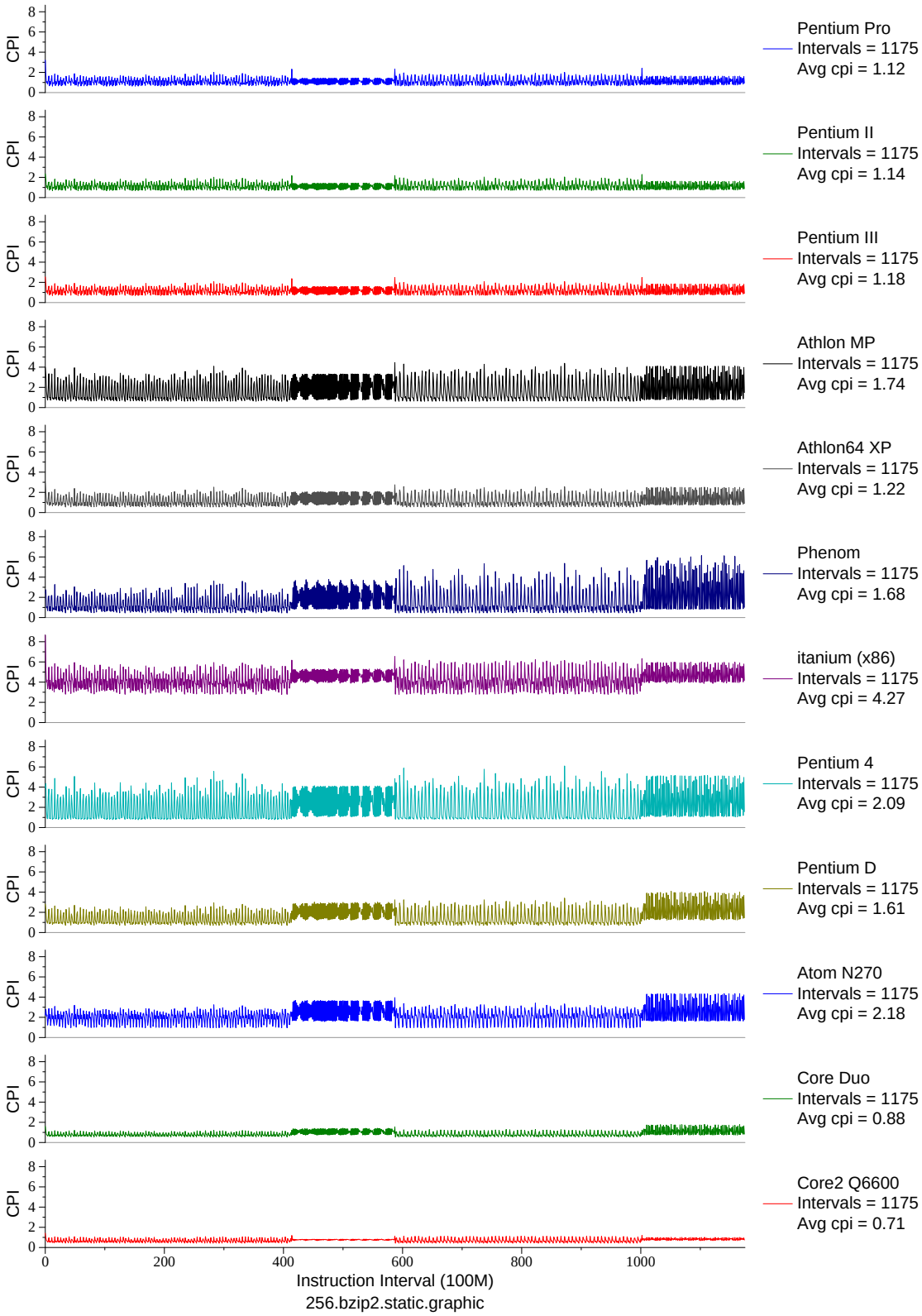

Figure E.44: CPI phase plot for `bzip2.graph` (INT, C, Compression)

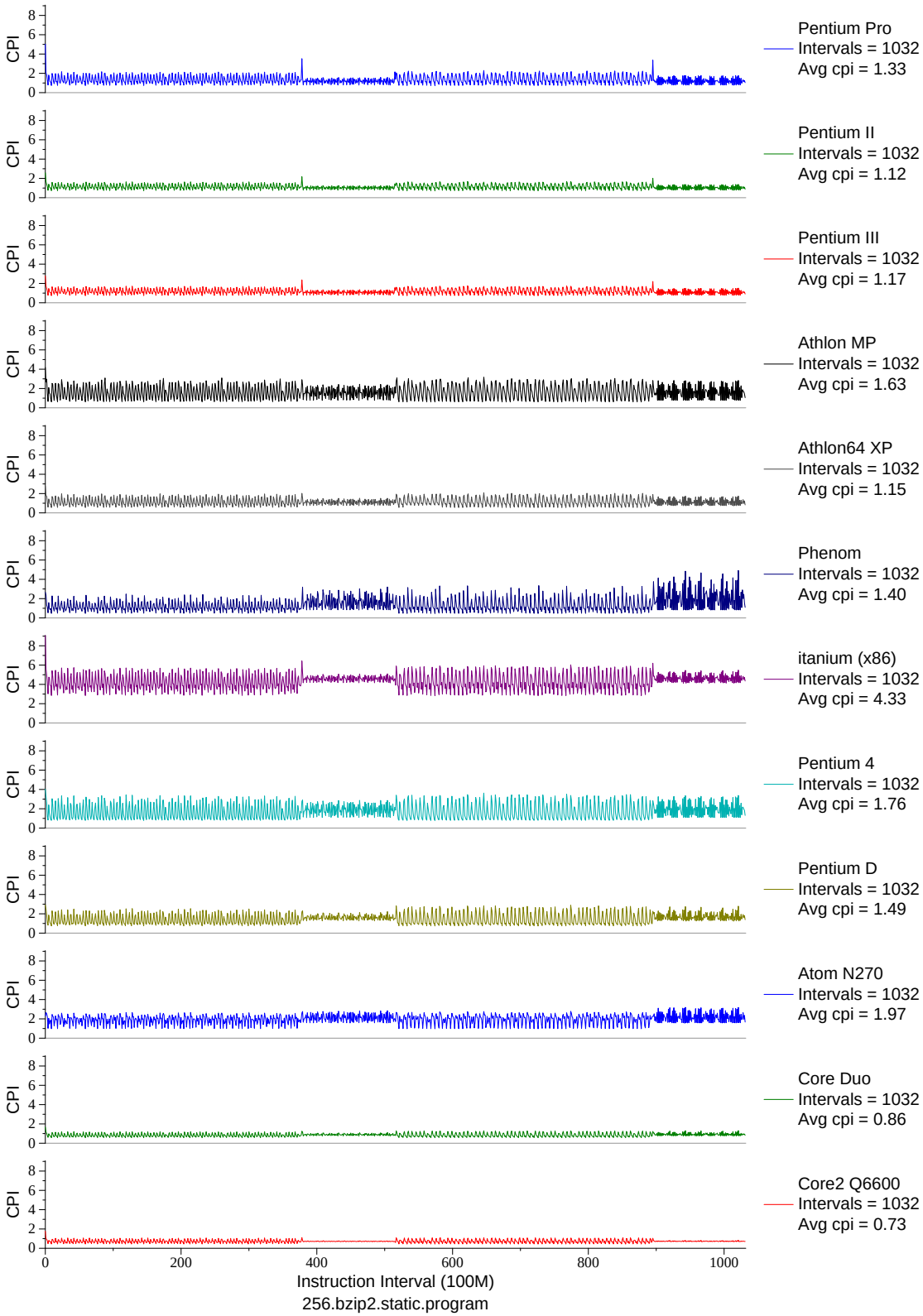

Figure E.45: CPI phase plot for `bzip2.prog` (INT, C, Compression)

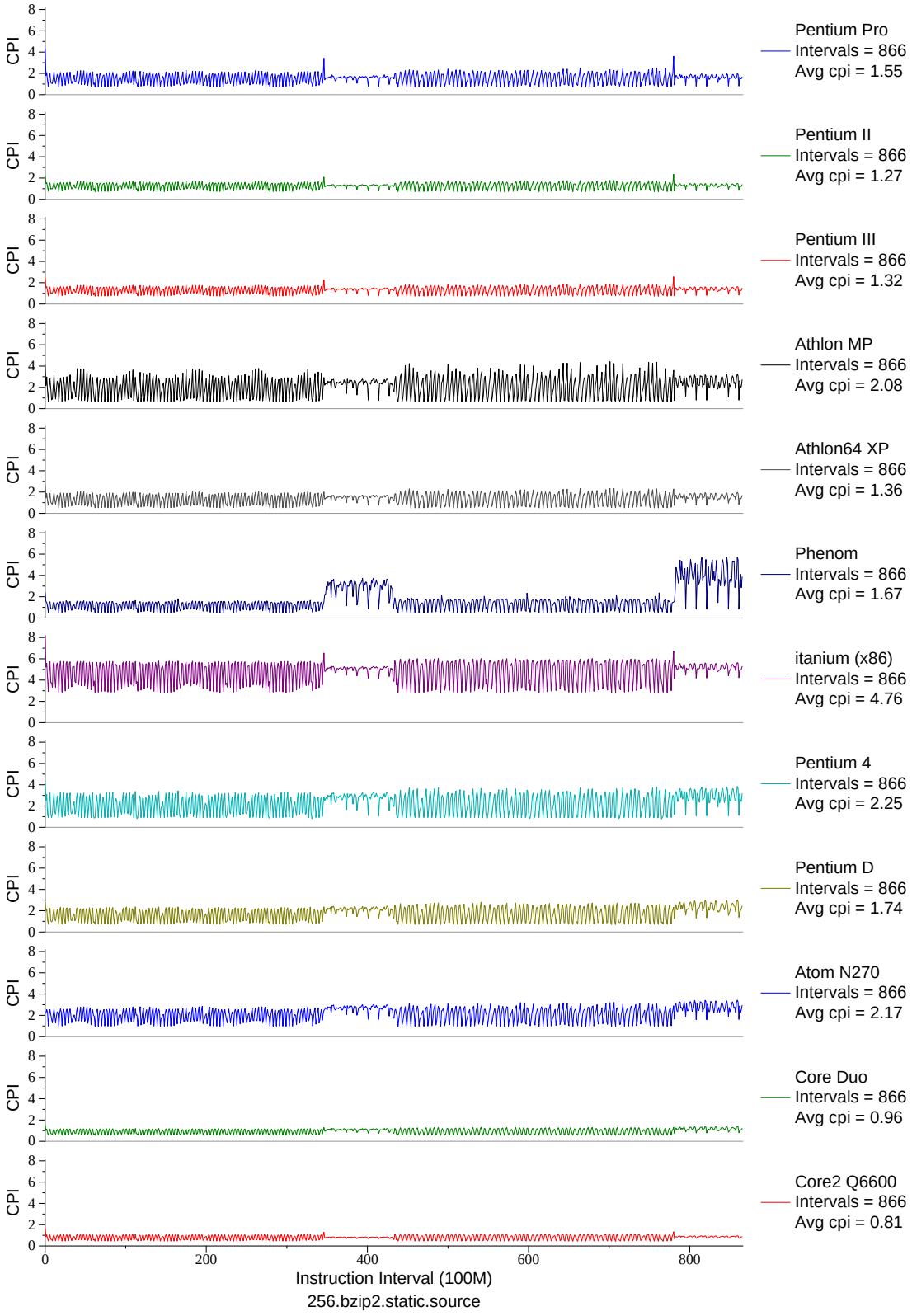

Figure E.46: CPI phase plot for `bzip2.src` (INT, C, Compression)

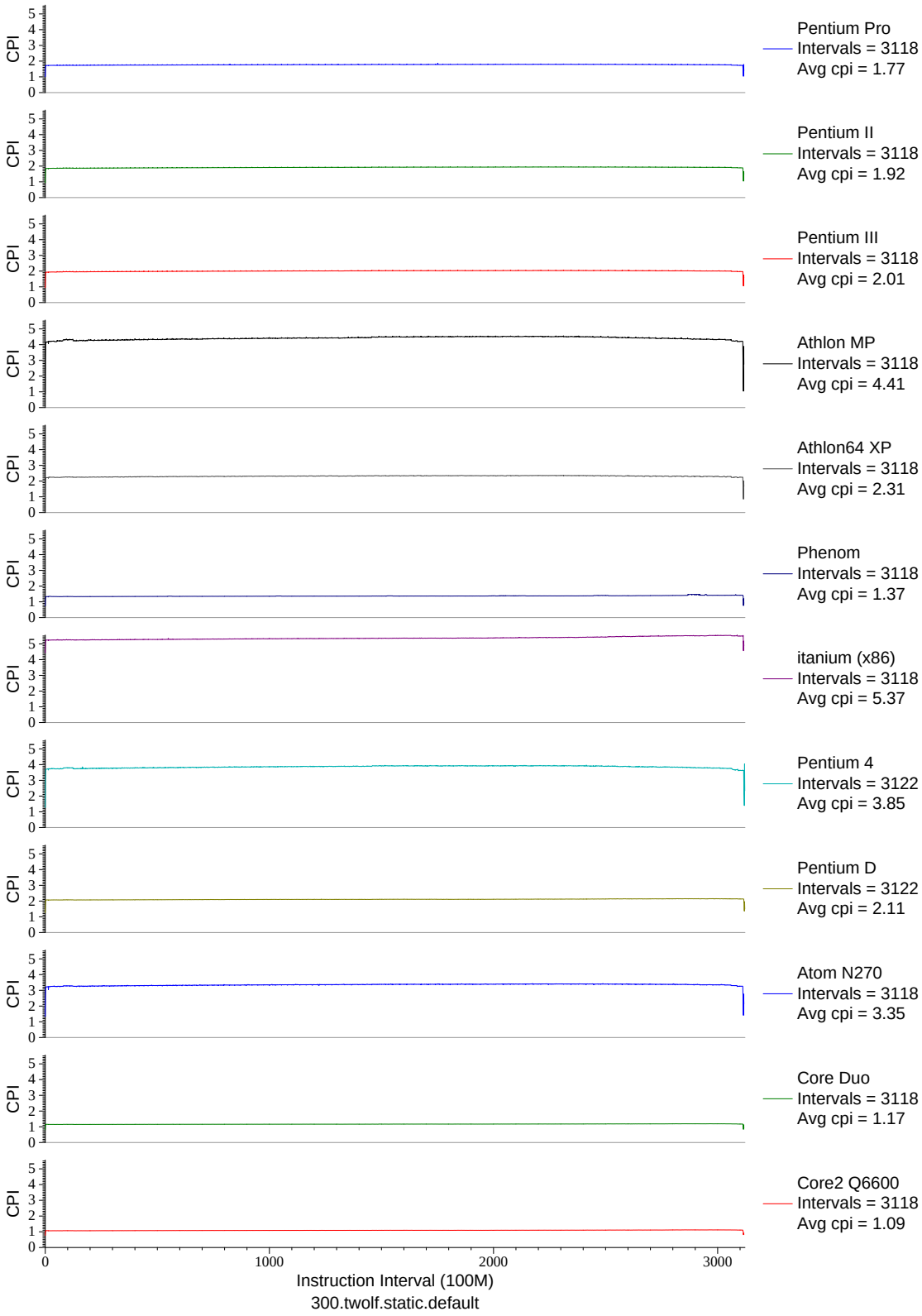

Figure E.47: CPI phase plot for `twolf` (INT, C, Place/Route)

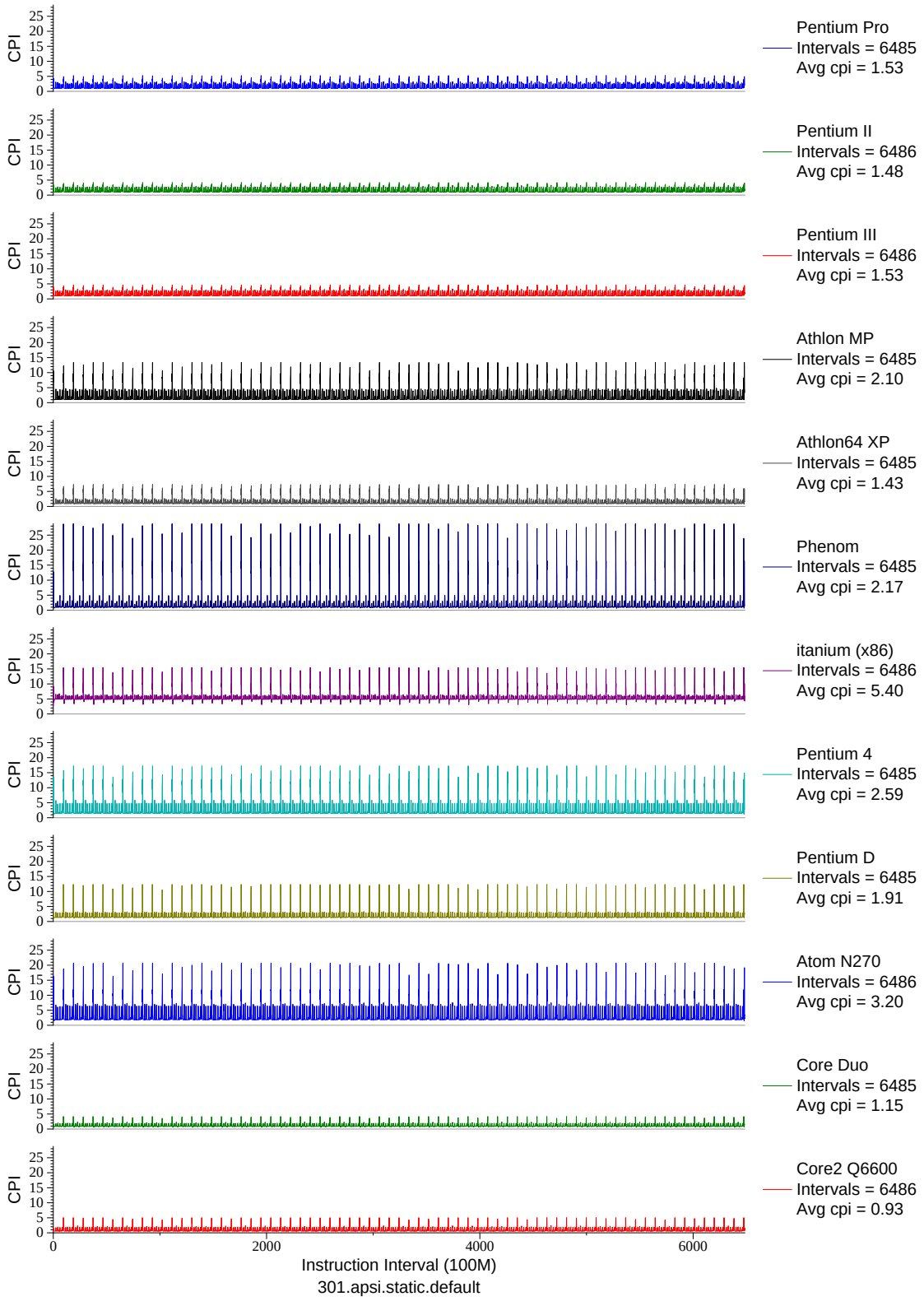

Figure E.48: CPI phase plot for `apsi` (FP, F77, Meteorology/Pollution)

E.2 64-bit x86_64

Due to issues inherent in the benchmarks themselves, there are no plots for the `vortex` benchmarks, nor the `perlbmk.535`, `perlbmk.704`, `perlbmk.850`, `perlbmk.957` or `perlbmk.diffmail`.

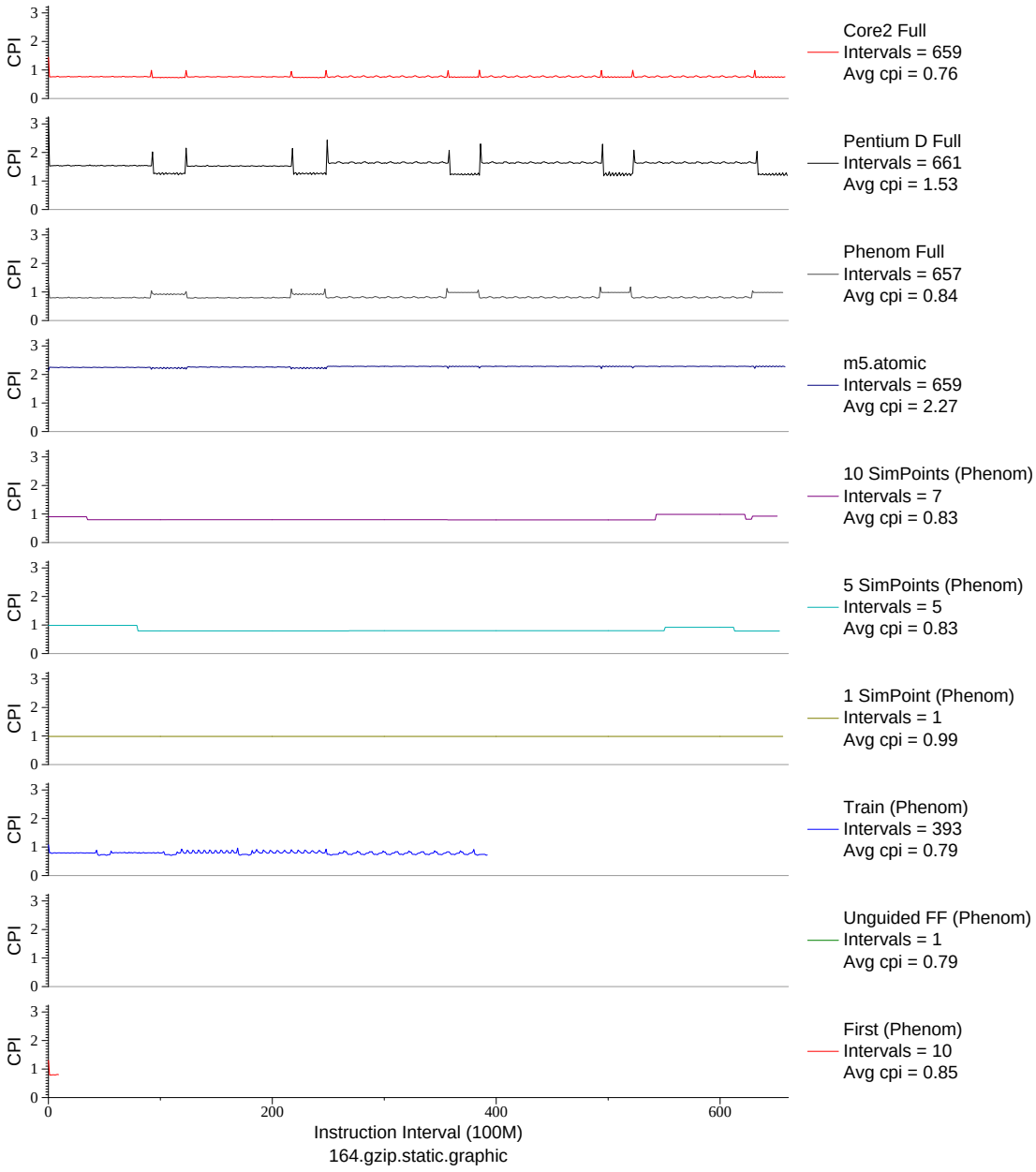

Figure E.49: CPI phase plot for `gzip.graph` (INT, C, Compression)

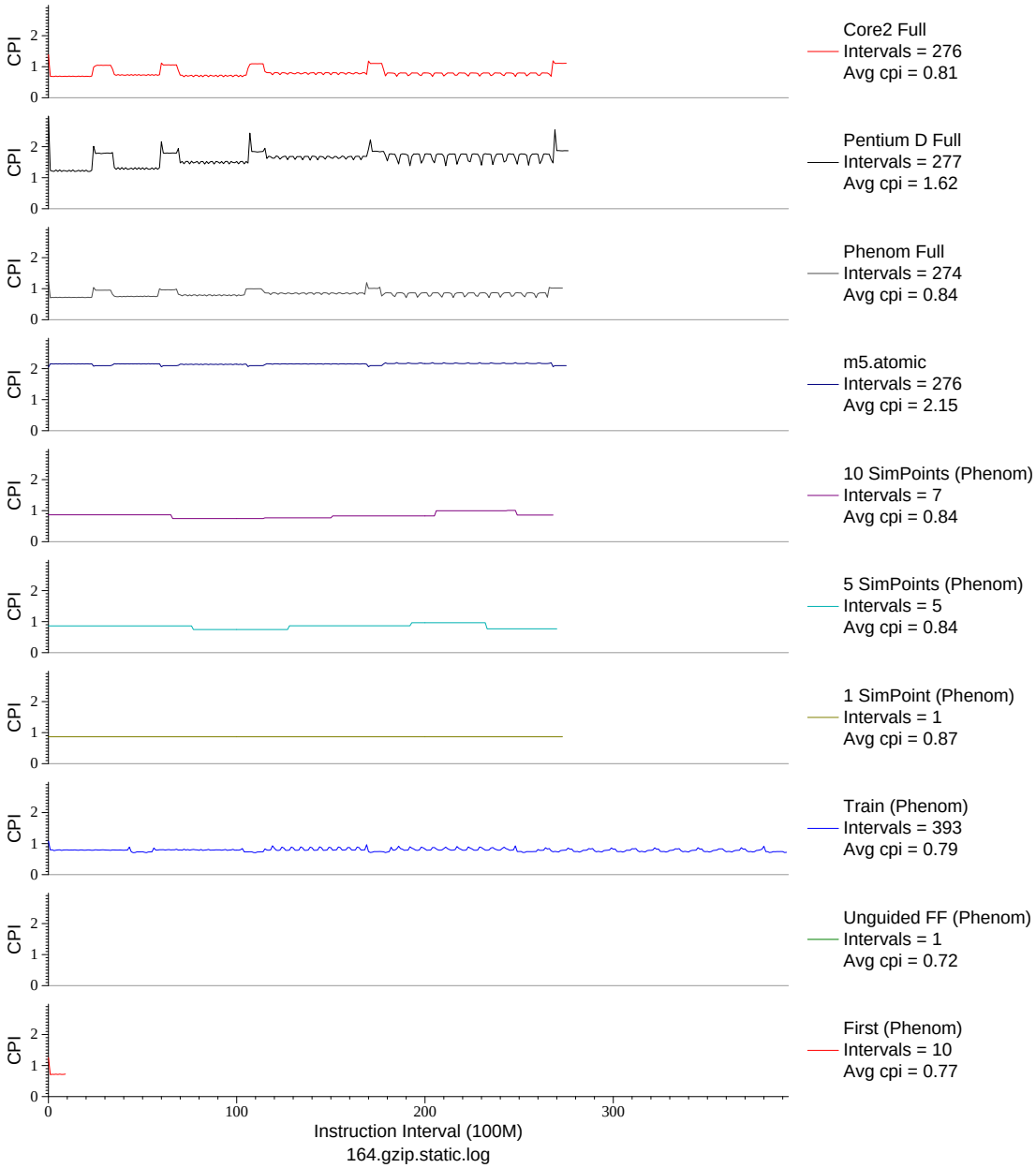

Figure E.50: CPI phase plot for `gzip.log` (INT, C, Compression)

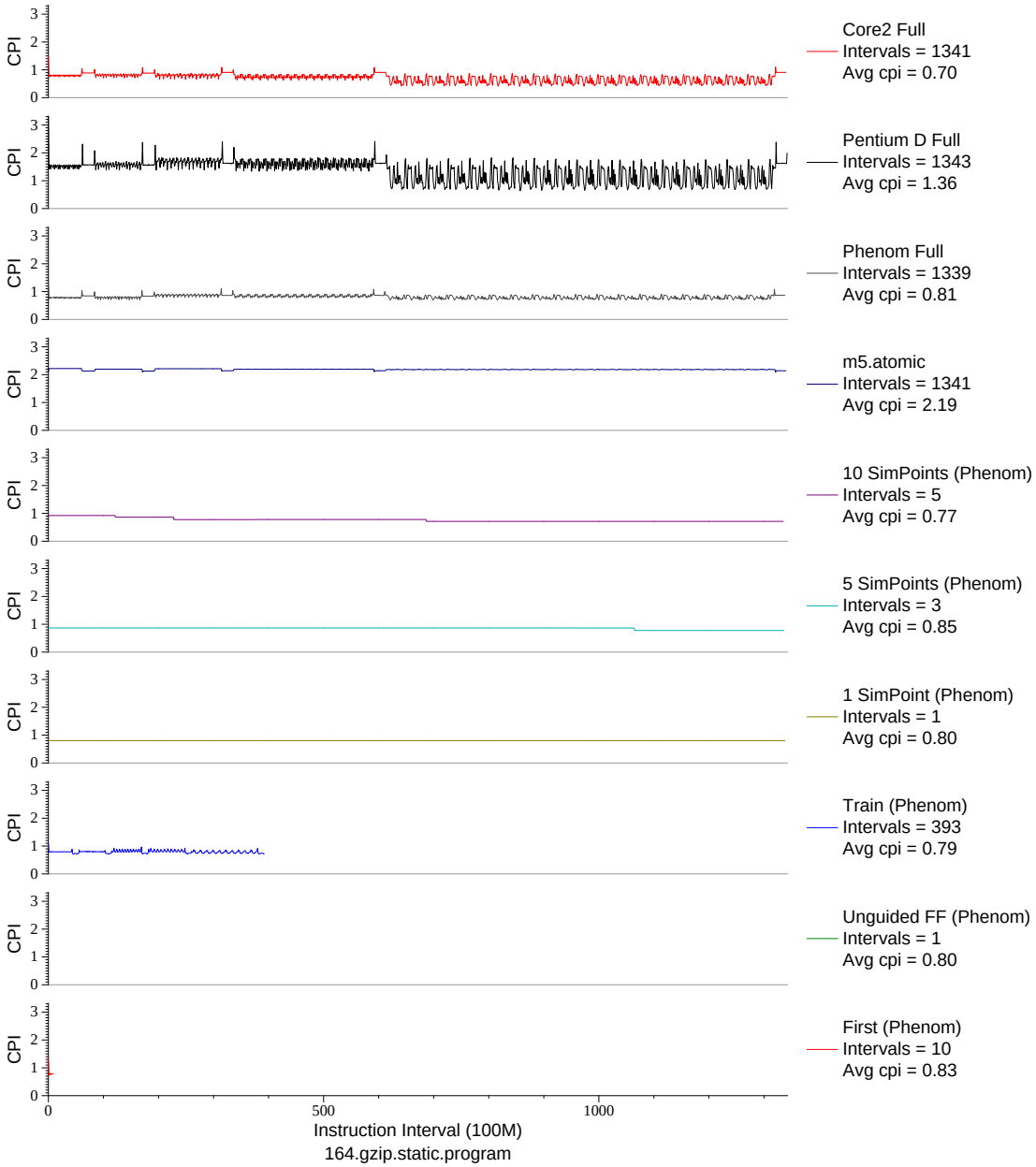

Figure E.51: CPI phase plot for `gzip.prog` (INT, C, Compression)

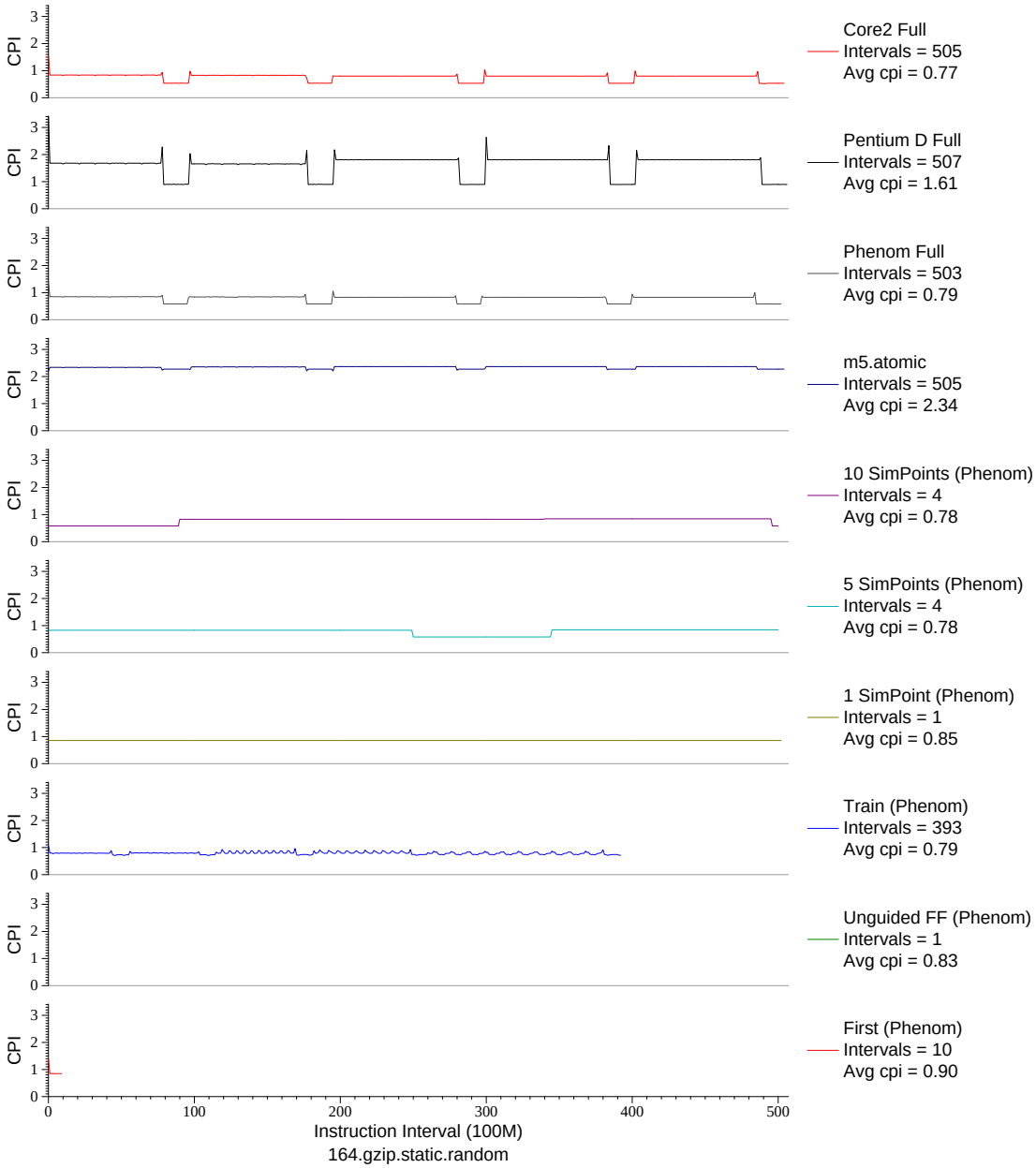

Figure E.52: CPI phase plot for `gzip.rnd` (INT, C, Compression)

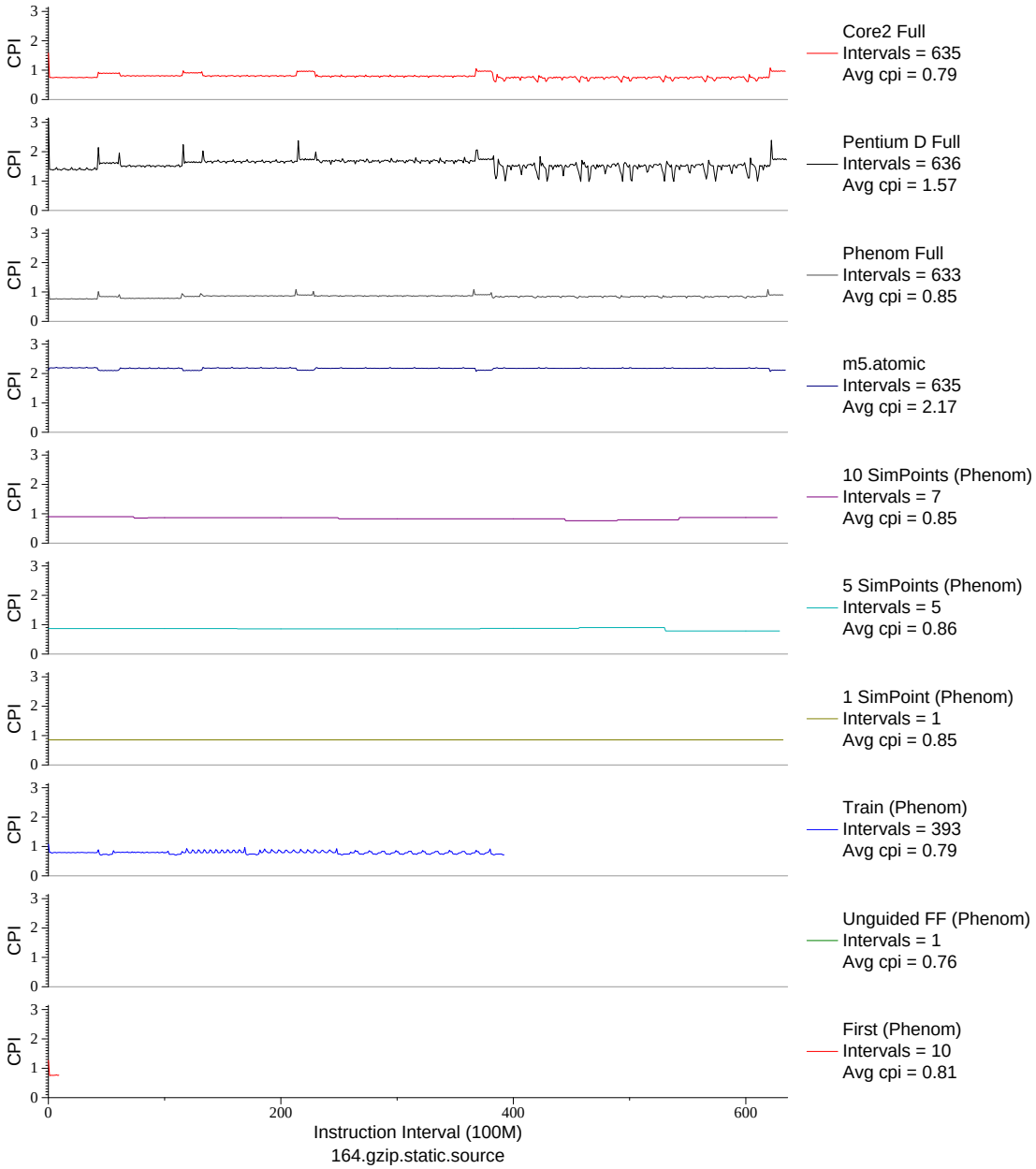

Figure E.53: CPI phase plot for `gzip.src` (INT, C, Compression)

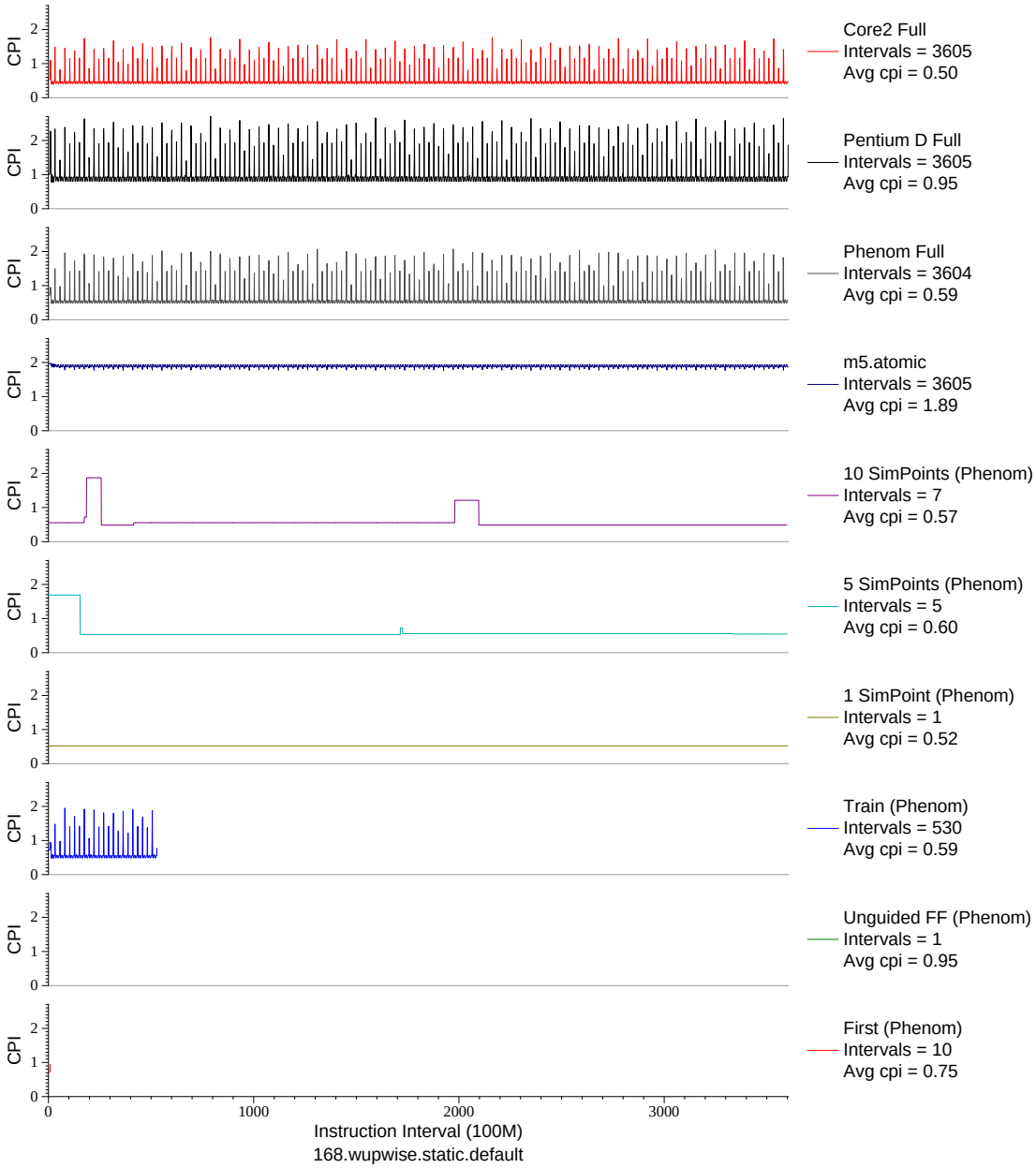

Figure E.54: CPI phase plot for `wupwise` (FP, F77, Quantum Chromodynamics)

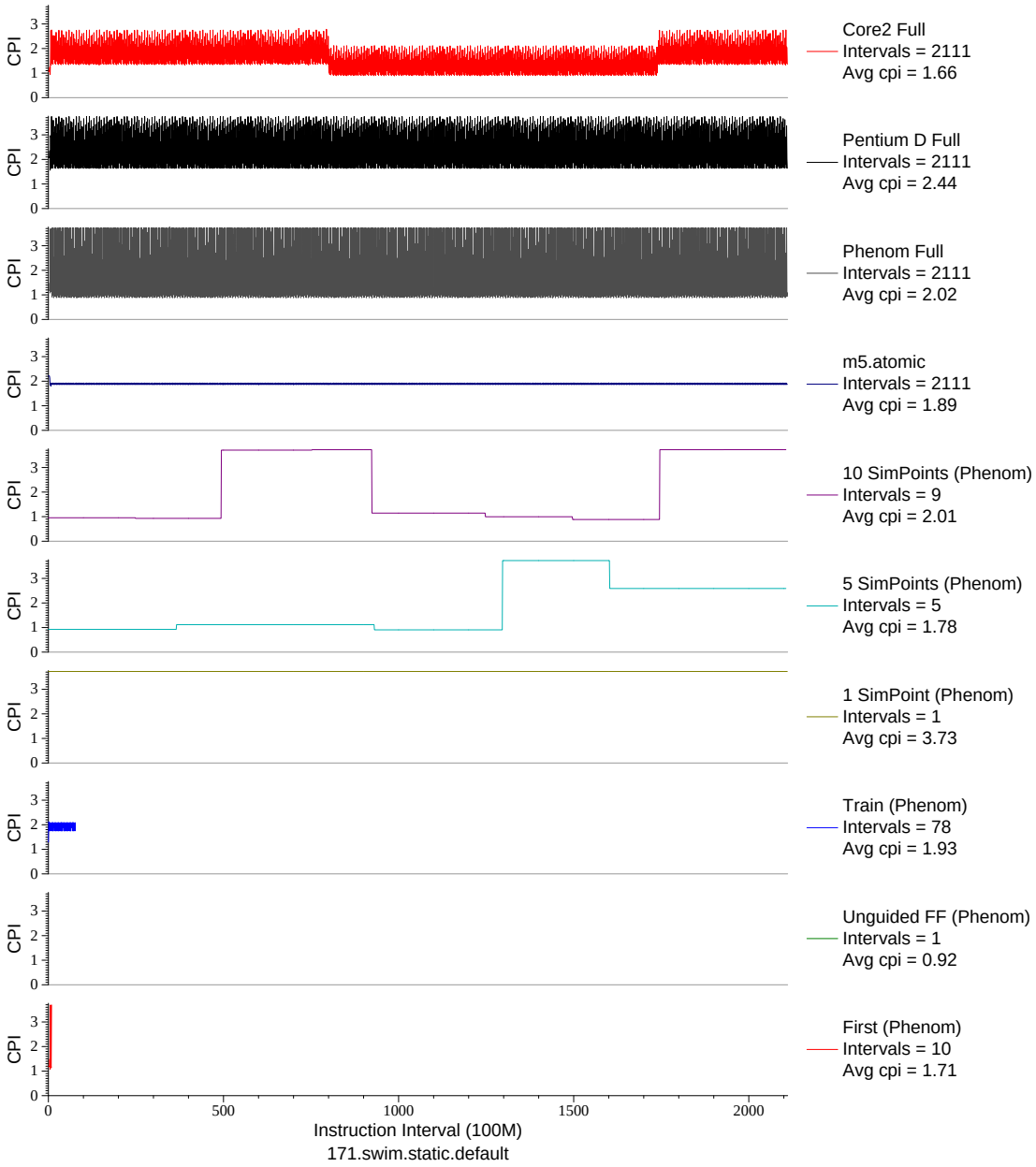

Figure E.55: CPI phase plot for `swim` (FP, F77, Meteorology/Water)

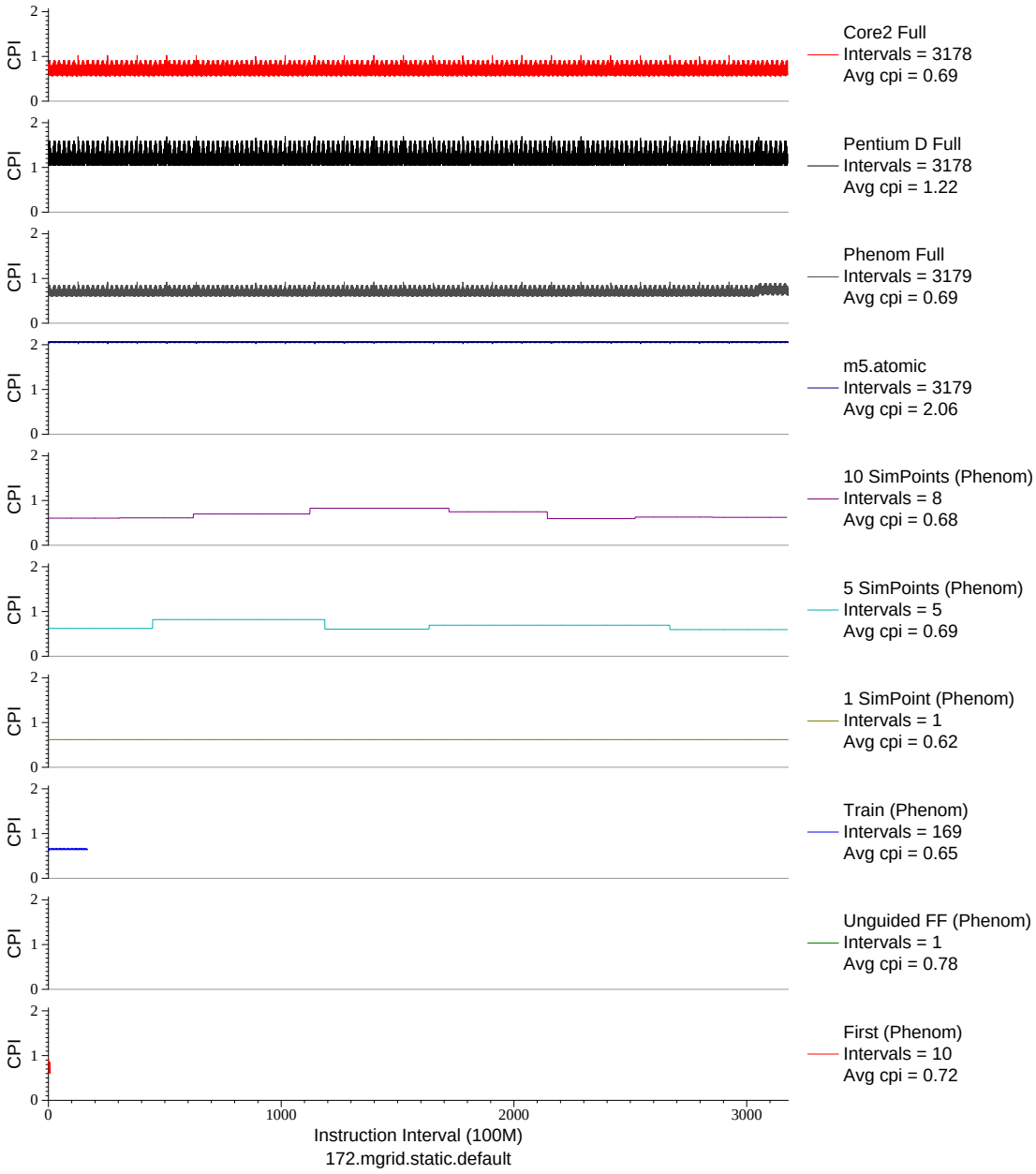

Figure E.56: CPI phase plot for `mgrid` (FP, F77, Multi-Grid Solver)

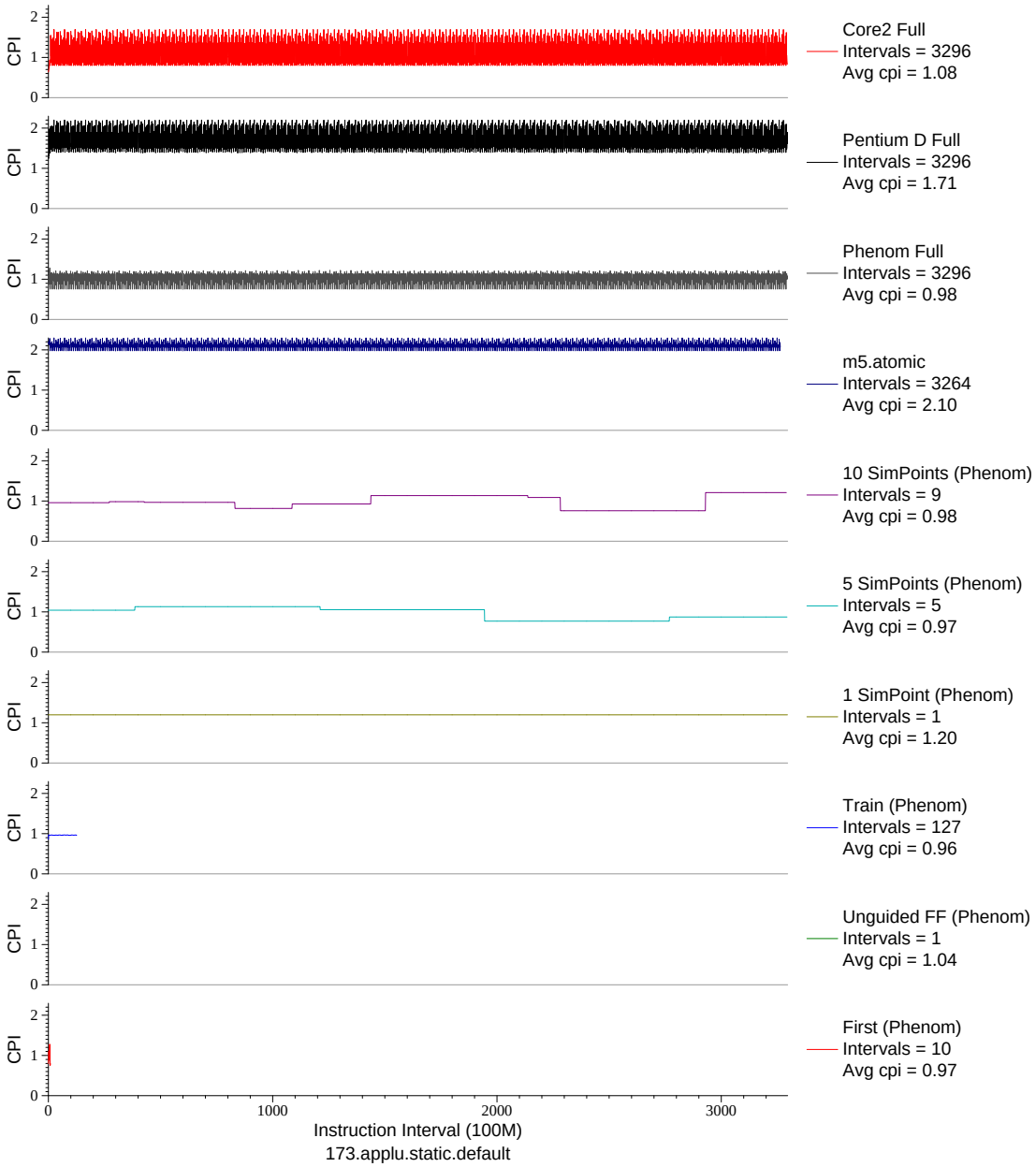

Figure E.57: CPI phase plot for `applu` (FP, F77, Fluid Dynamics)

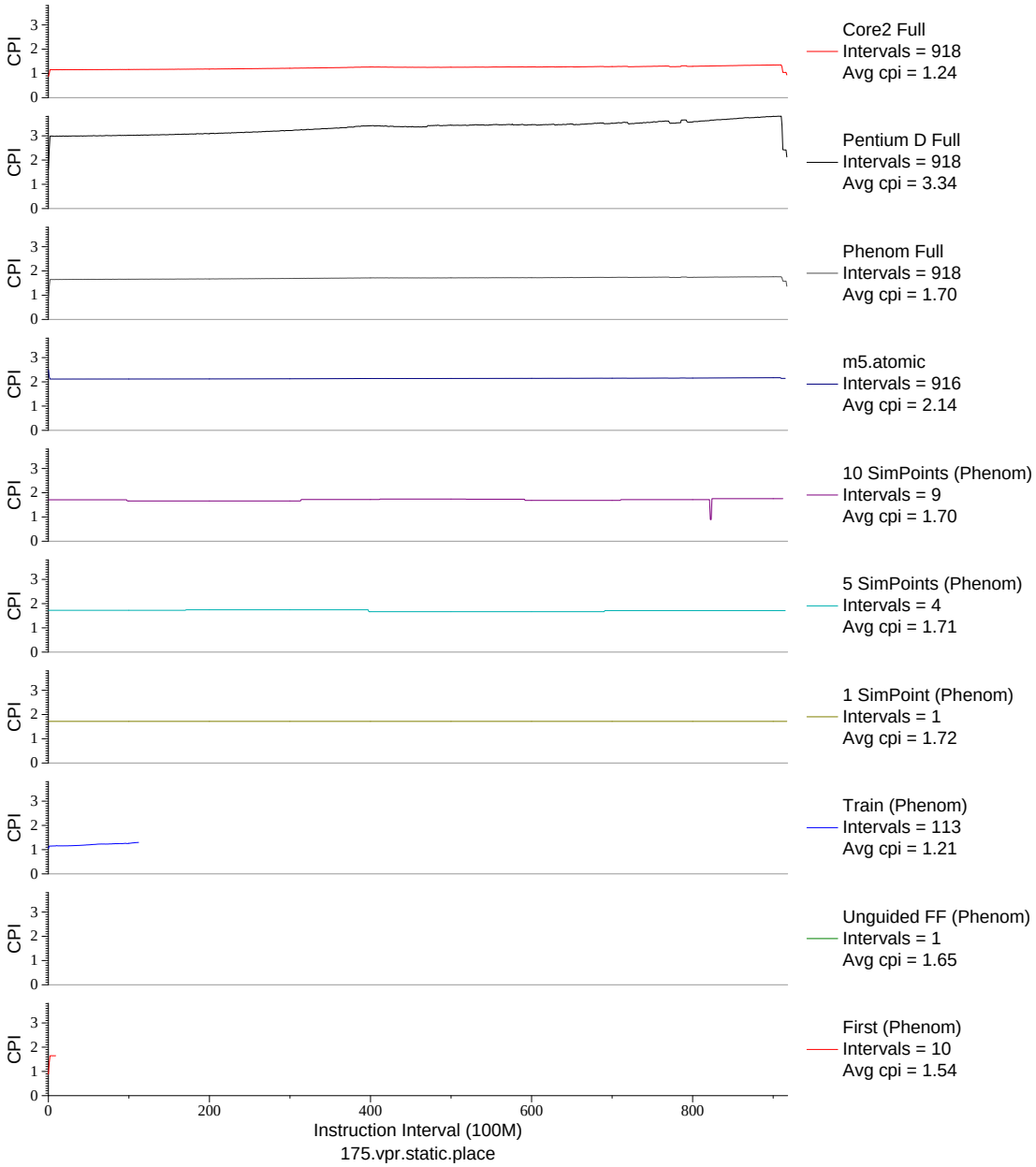

Figure E.58: CPI phase plot for `vpr.place` (INT, C, FPGA Place/Route)

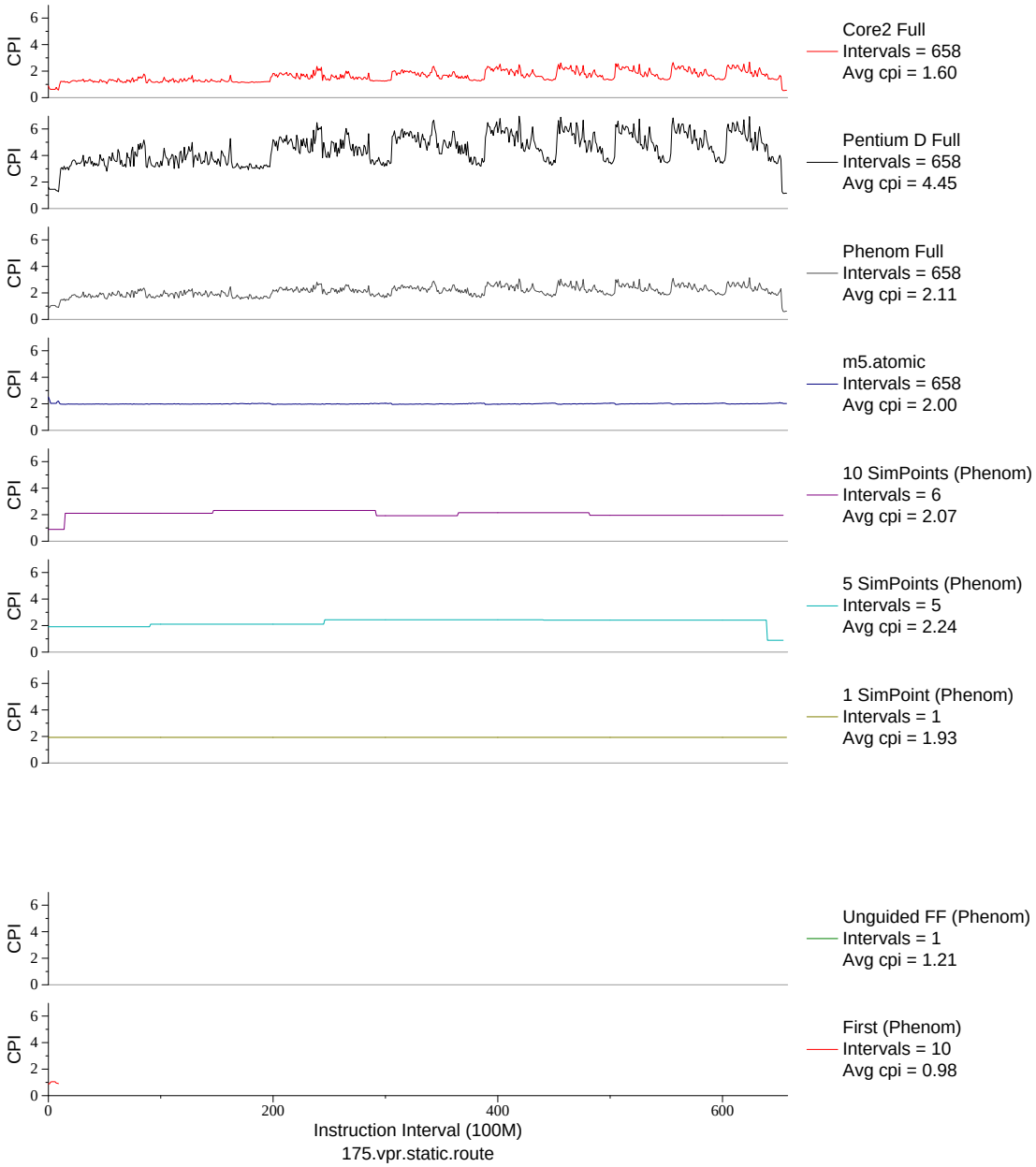

Figure E.59: CPI phase plot for `vpr.route` (INT, C, FPGA Place/Route)

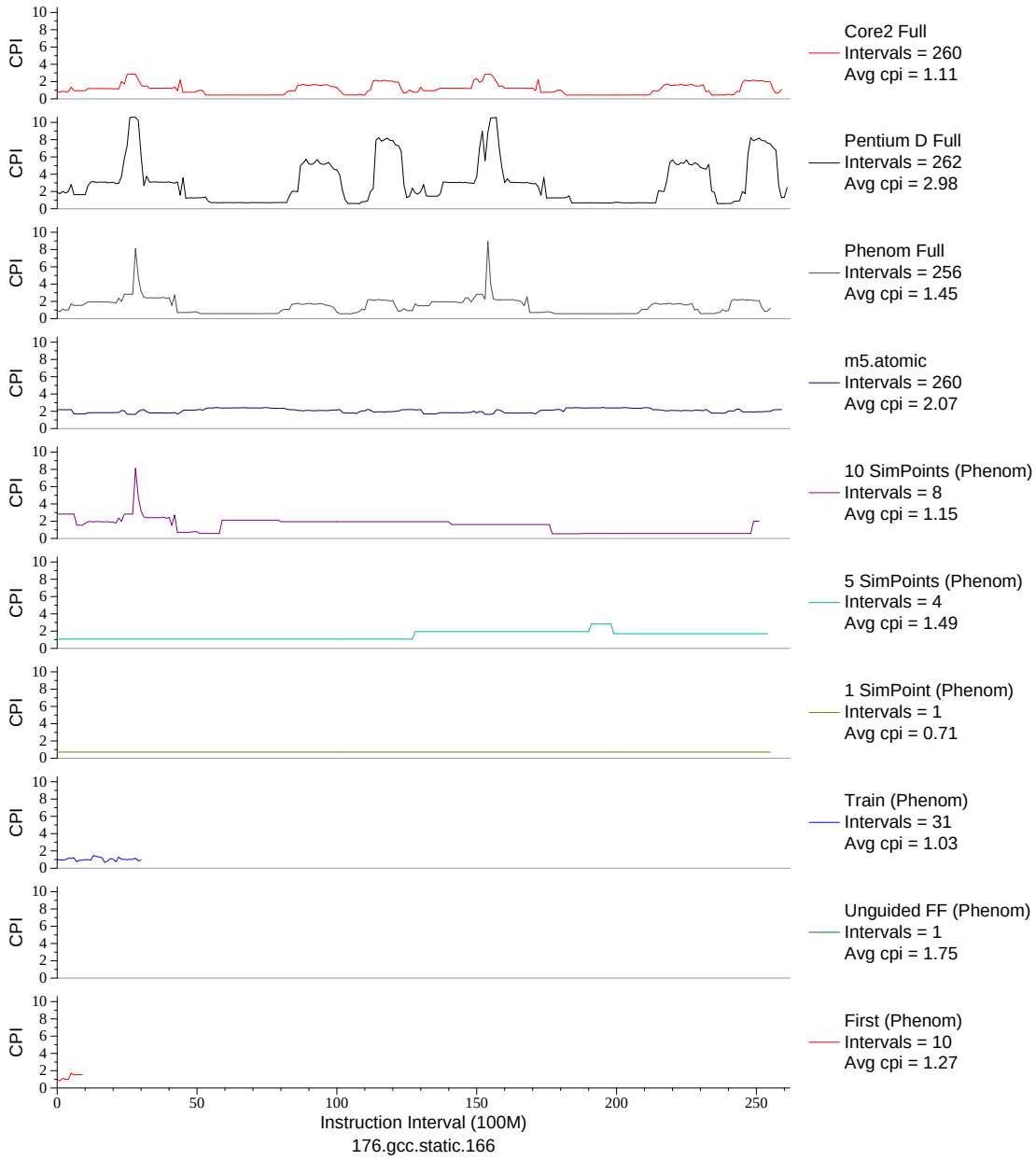

Figure E.60: CPI phase plot for `gcc.166` (INT, C, C Compiler)

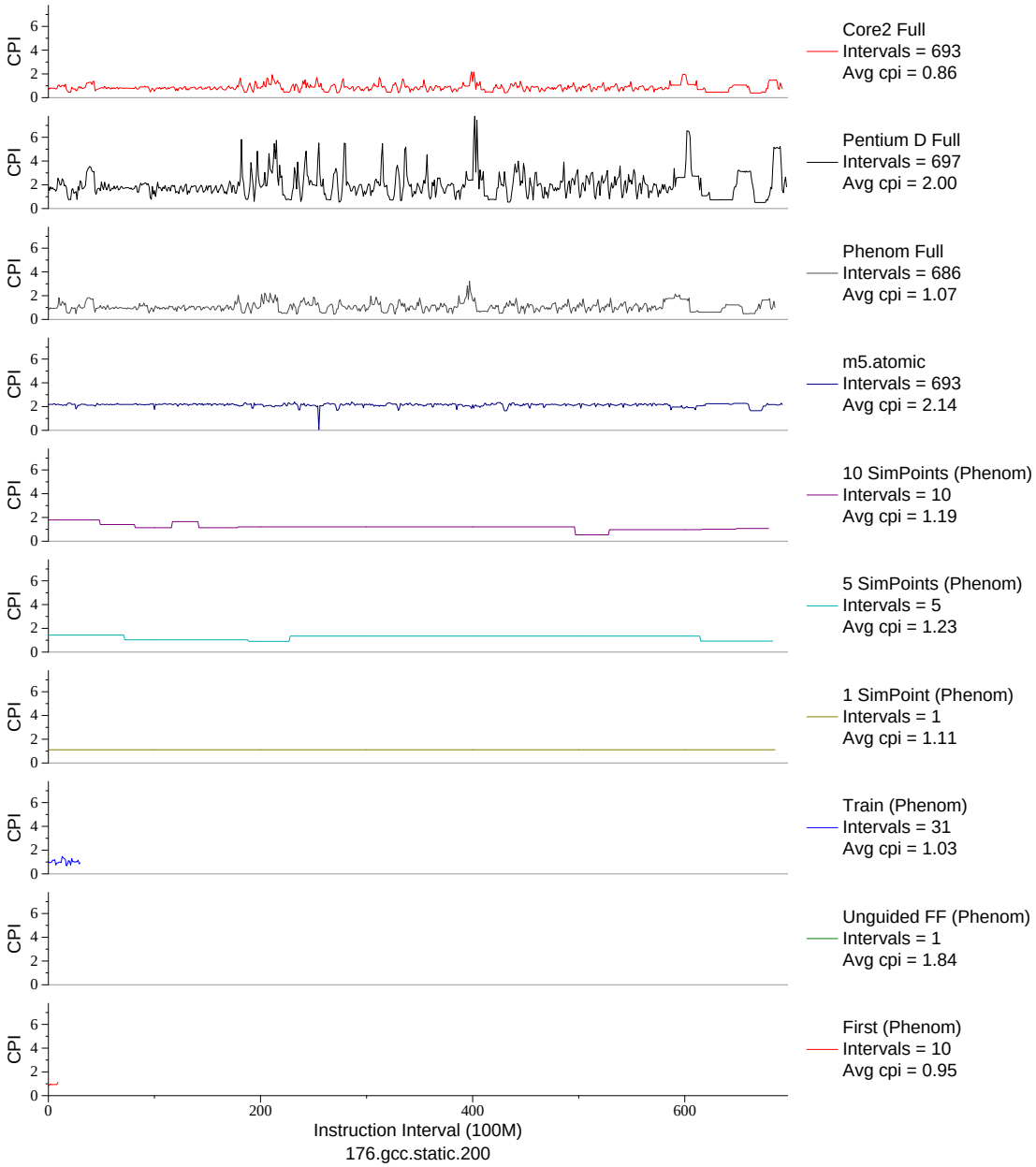

Figure E.61: CPI phase plot for `gcc.200` (INT, C, C Compiler)

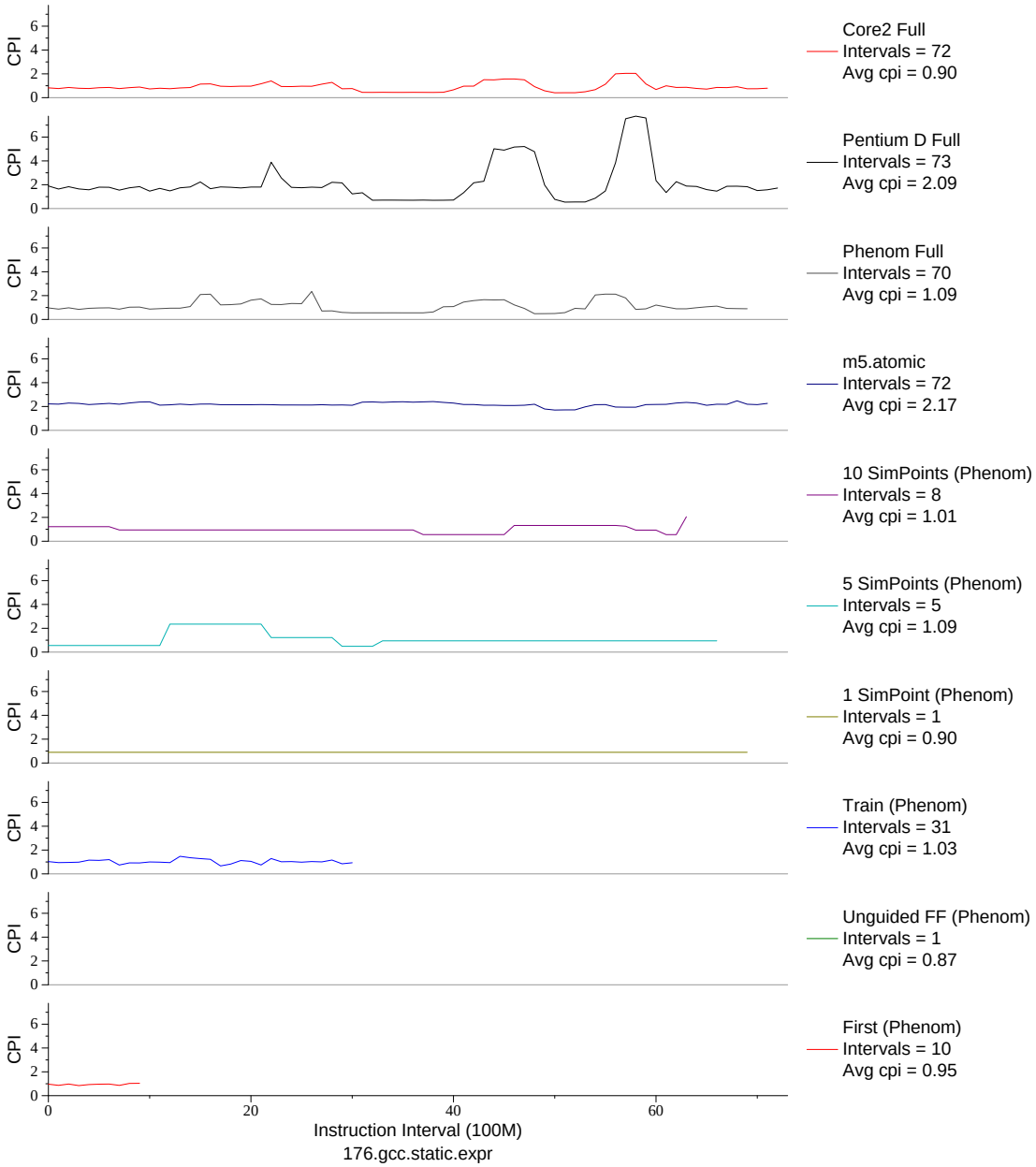

Figure E.62: CPI phase plot for `gcc.expr` (INT, C, C Compiler)

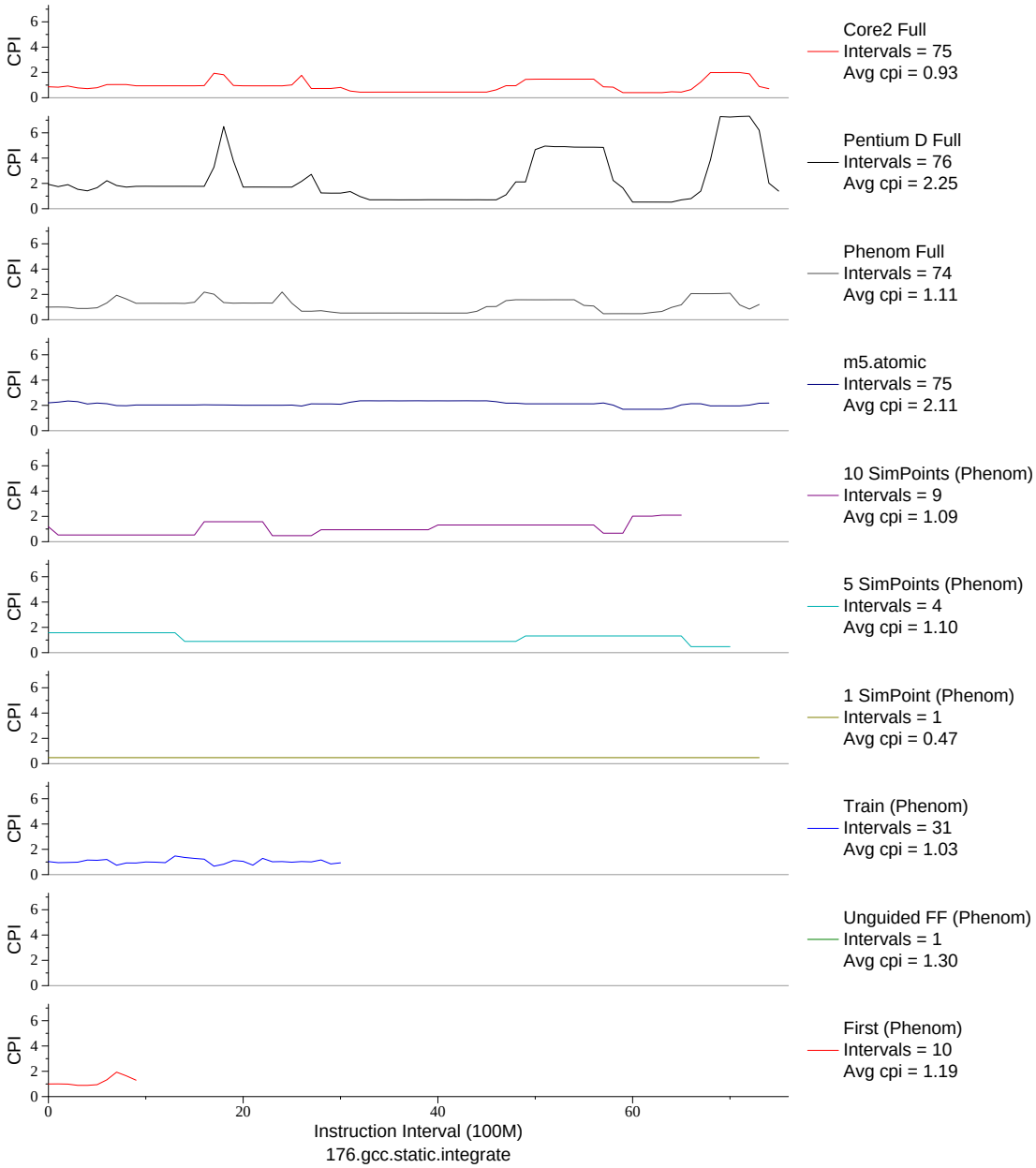

Figure E.63: CPI phase plot for `gcc.int` (INT, C, C Compiler)

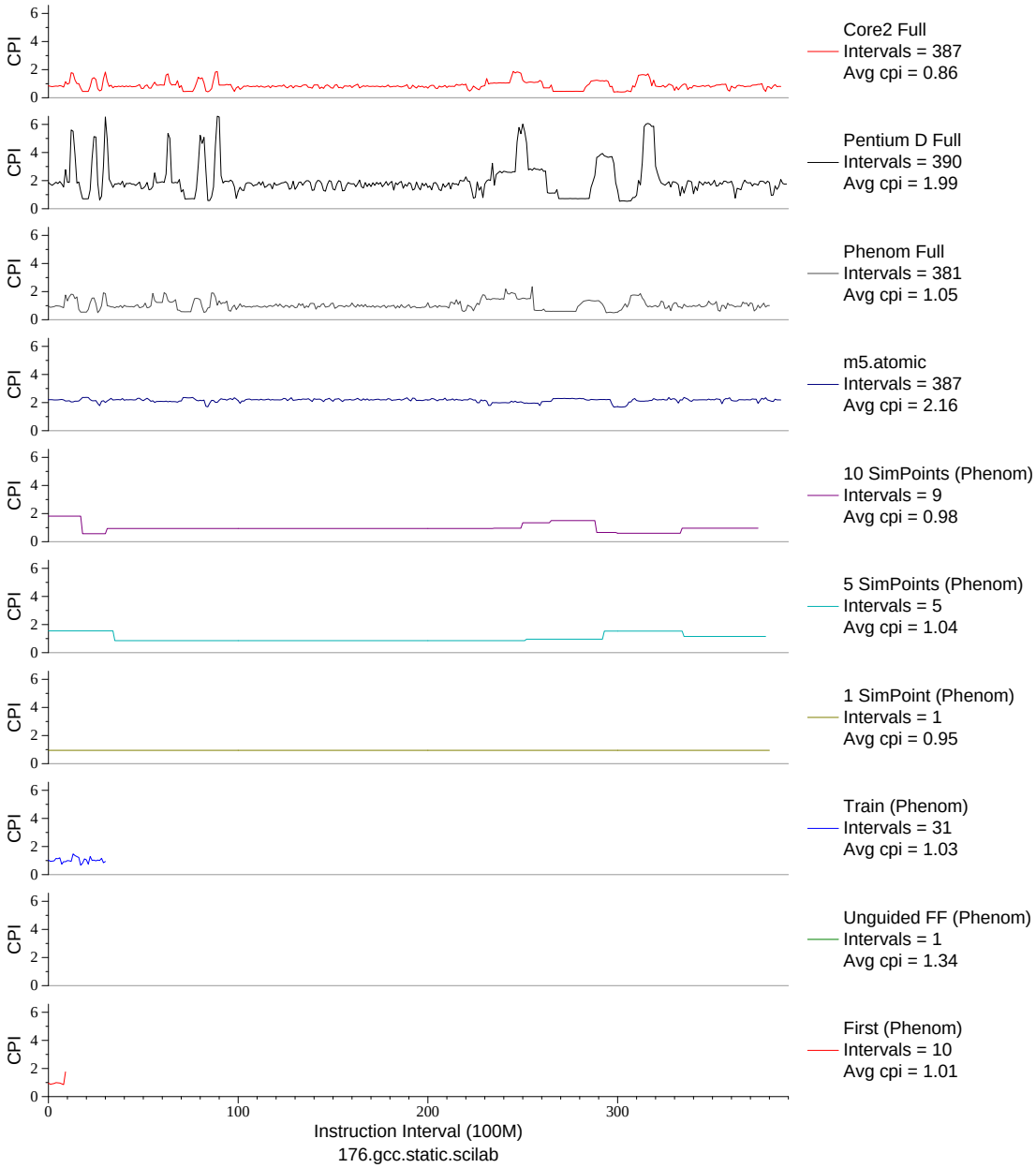

Figure E.64: CPI phase plot for `gcc.sci` (INT, C, C Compiler)

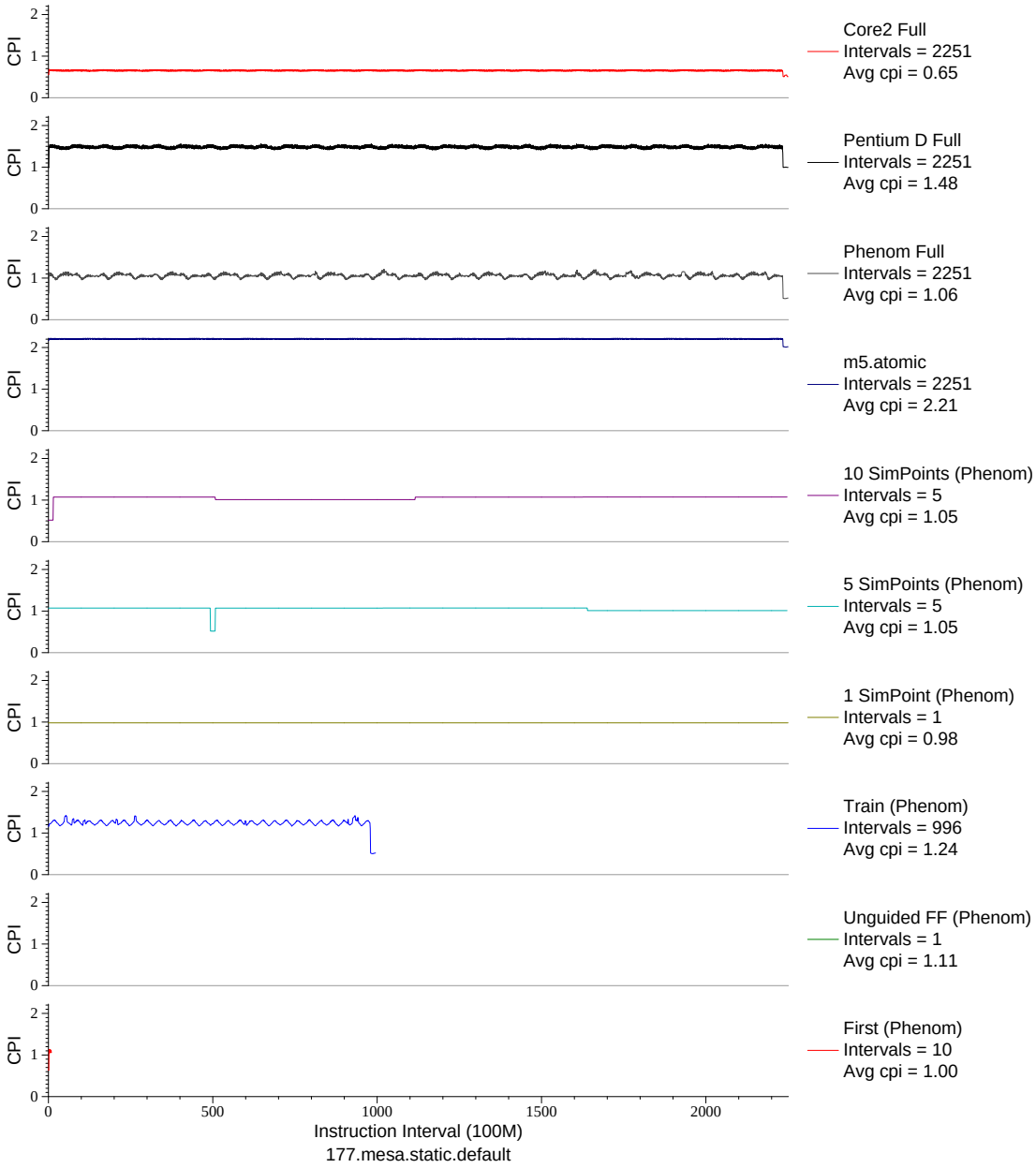

Figure E.65: CPI phase plot for `mesa` (FP, C, 3D-graphics)

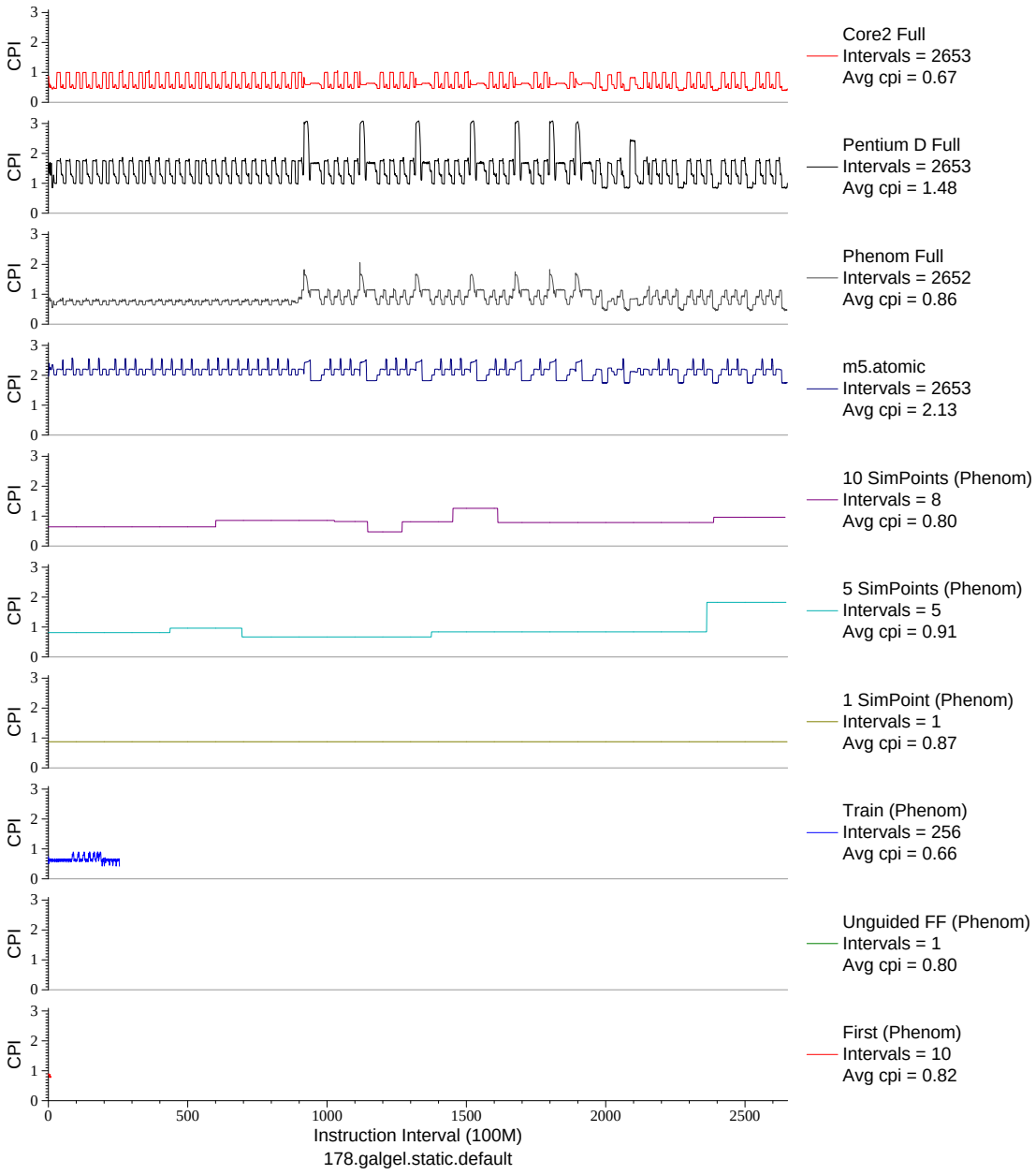

Figure E.66: CPI phase plot for `galgel` (FP, F90, Fluid Dynamics)

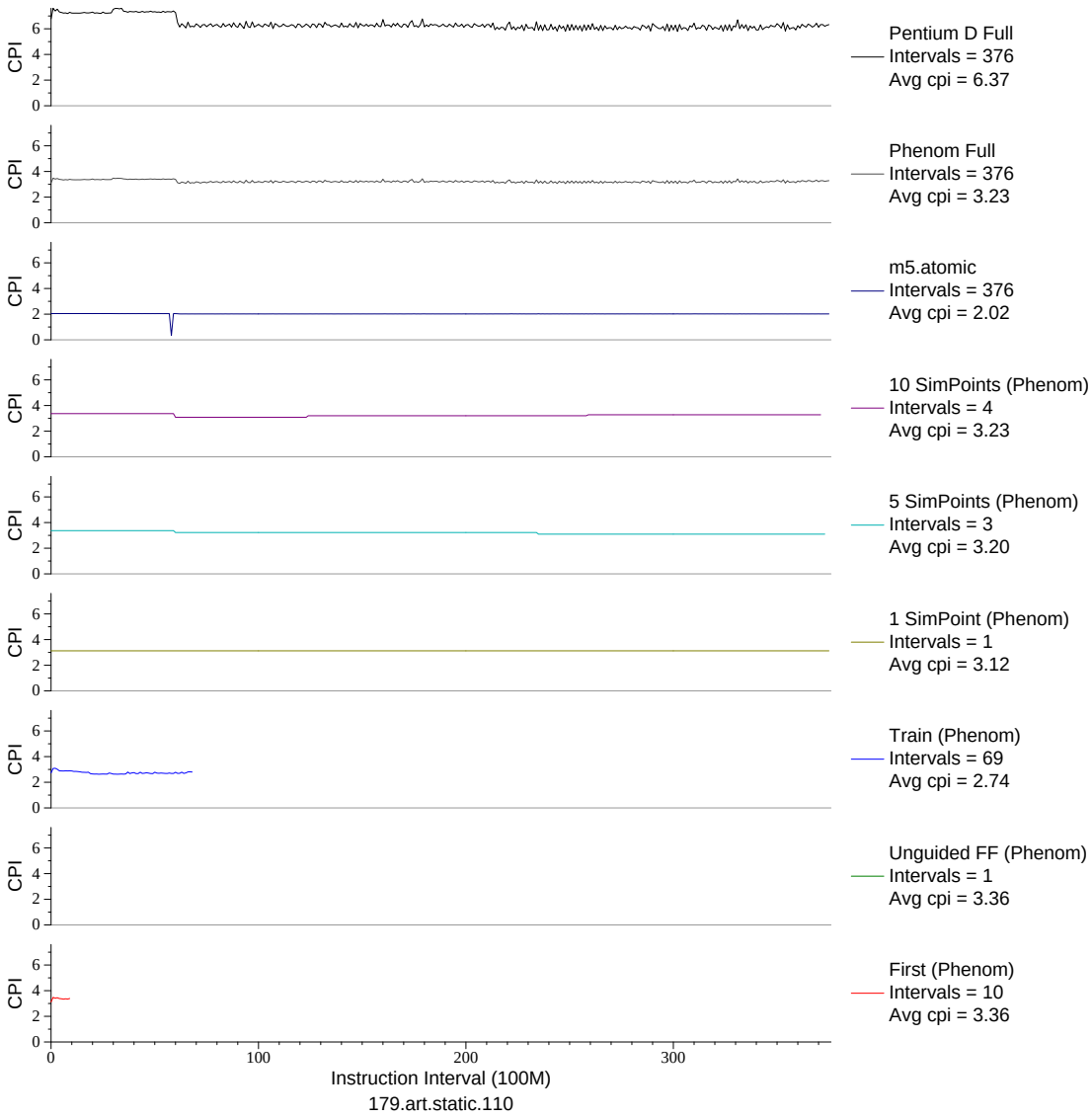

Figure E.67: CPI phase plot for `art.110` (FP, C, Neural Networks)

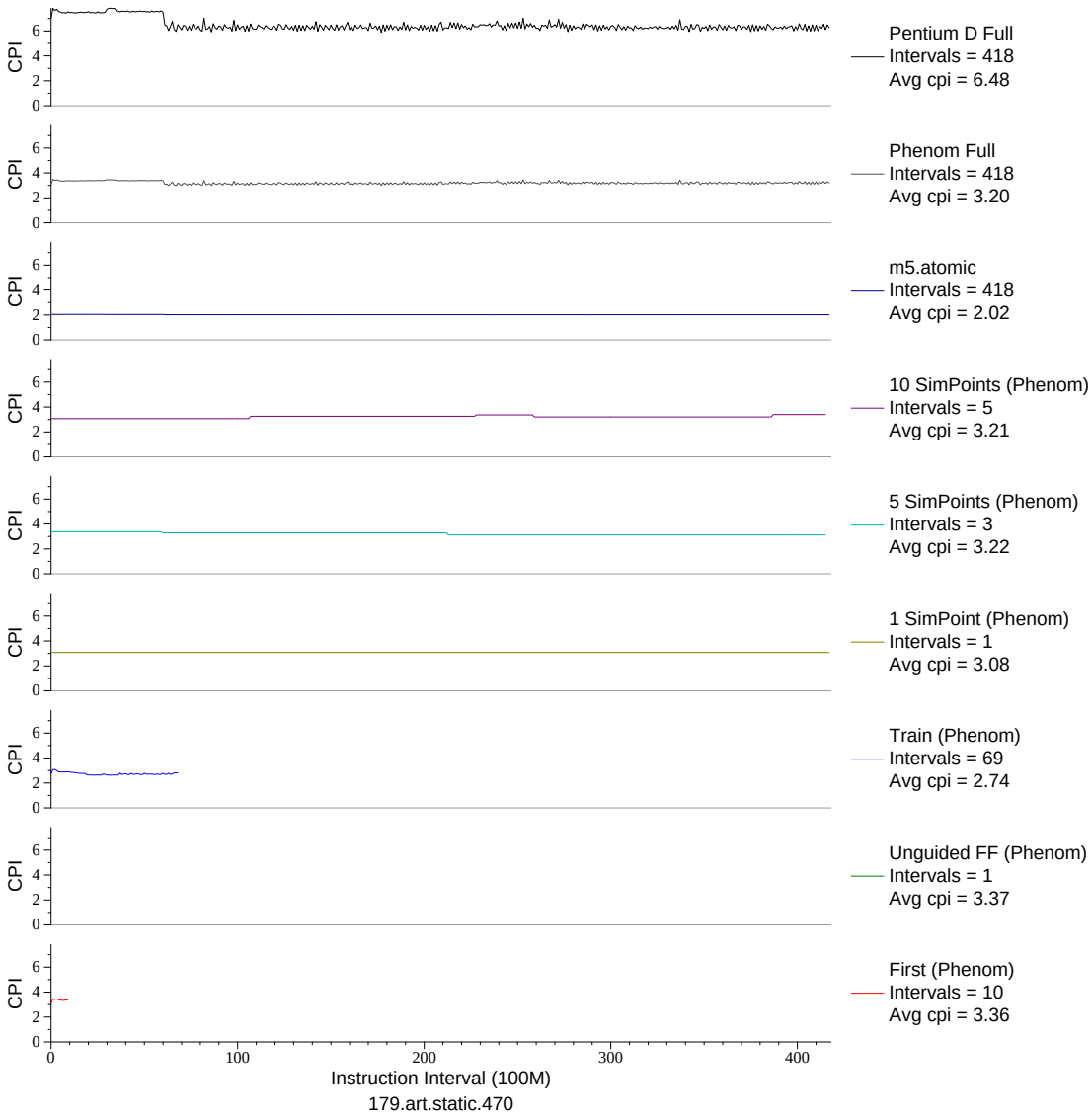

Figure E.68: CPI phase plot for `art.470` (FP, C, Neural Networks)

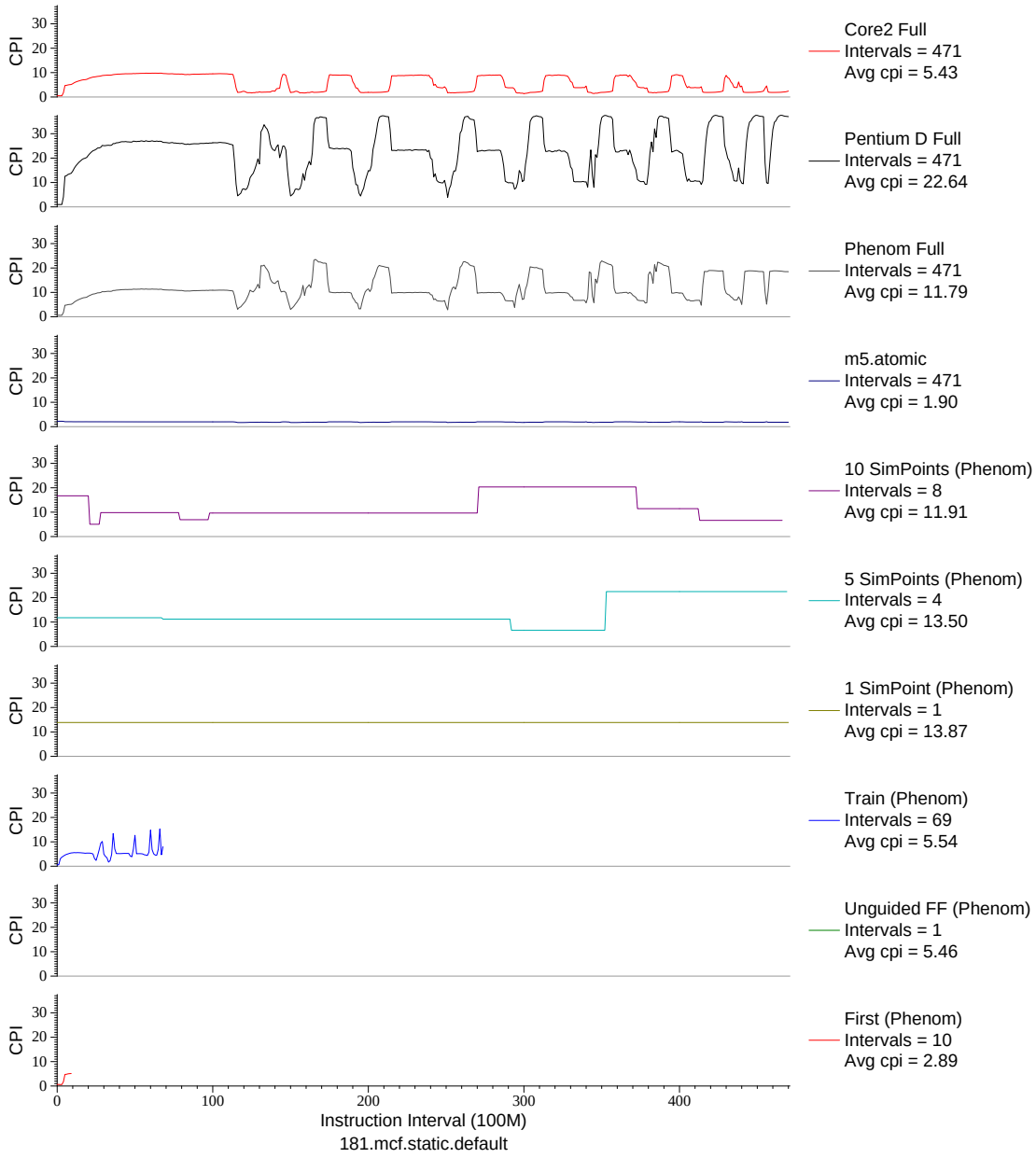

Figure E.69: CPI phase plot for `mcf` (INT, C, Combinatorial Opt)

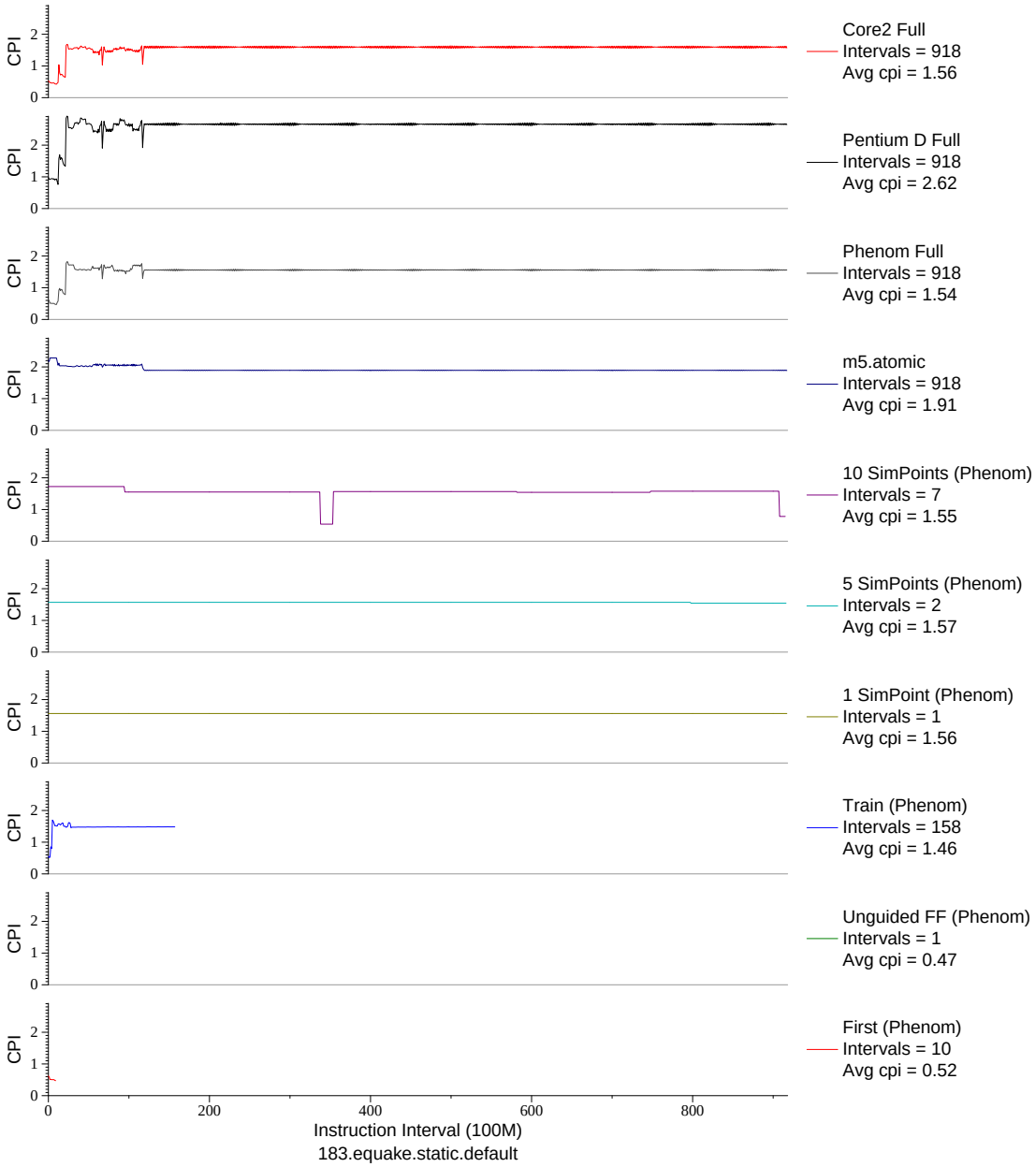

Figure E.70: CPI phase plot for `equake` (FP, C, Seismic Propogation)

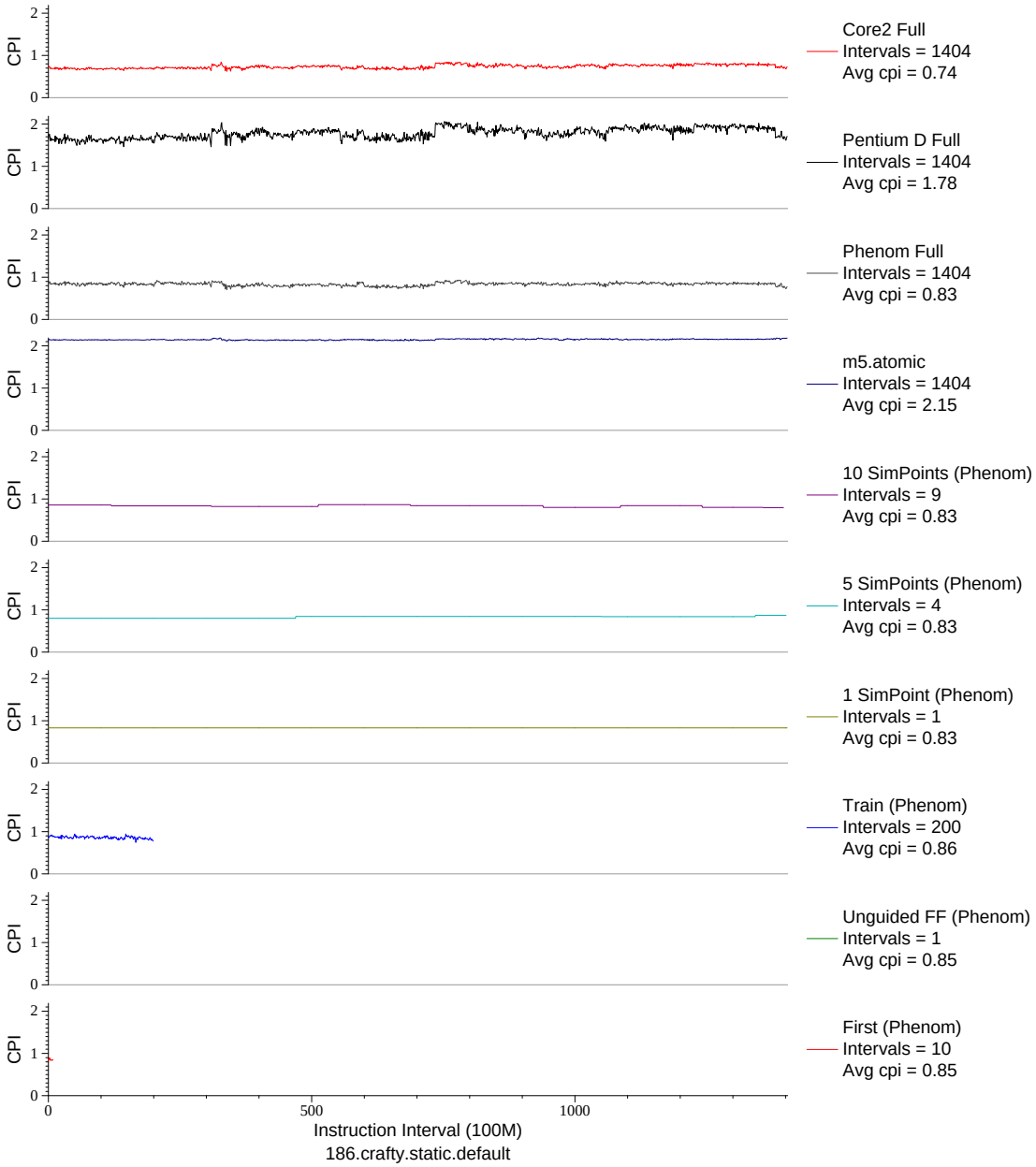

Figure E.71: CPI phase plot for `crafty` (INT, C, Chess)

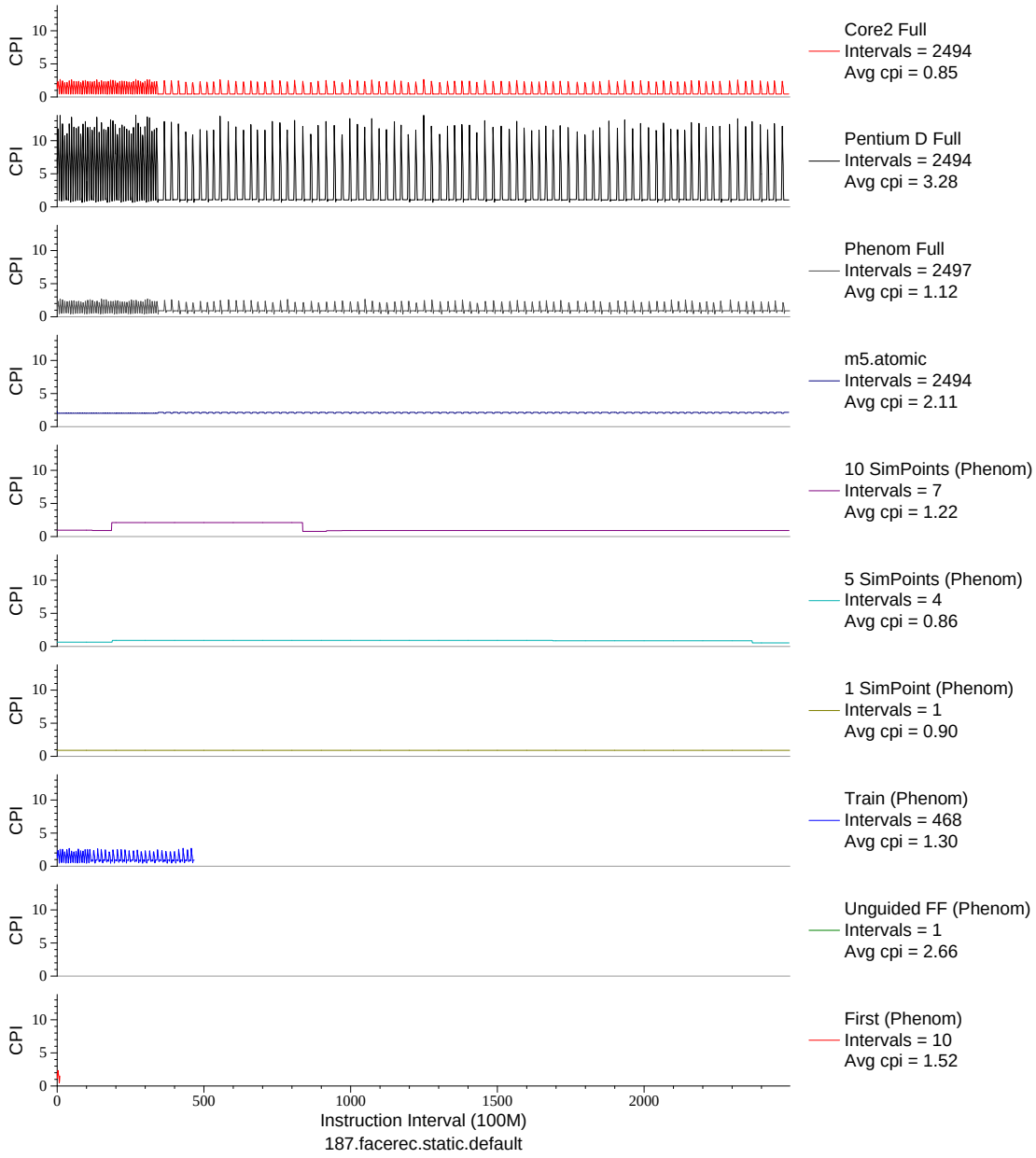

Figure E.72: CPI phase plot for `facerec` (FP, F90, Facial Recognition)

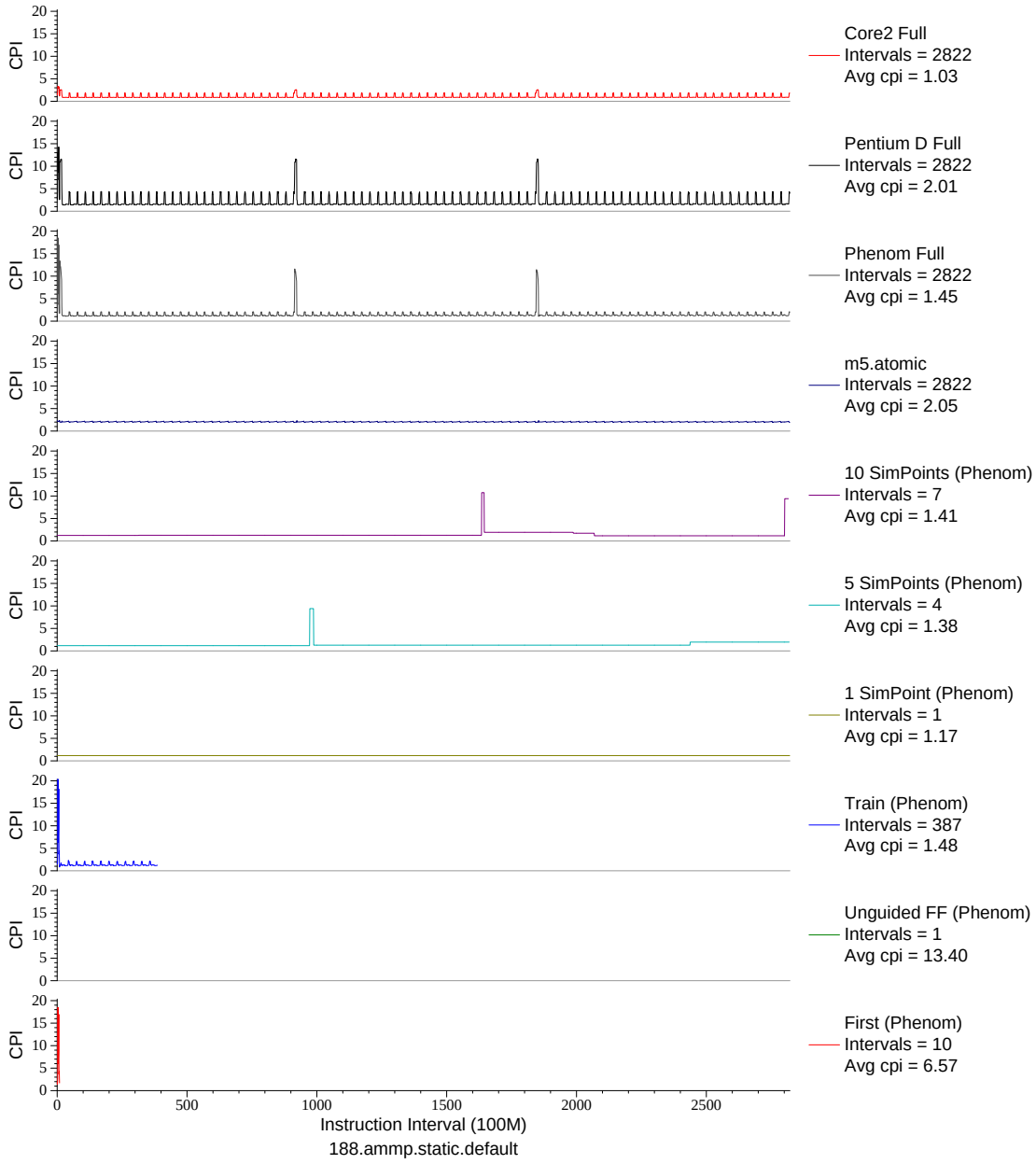

Figure E.73: CPI phase plot for `ammp` (FP, C, Chemistry)

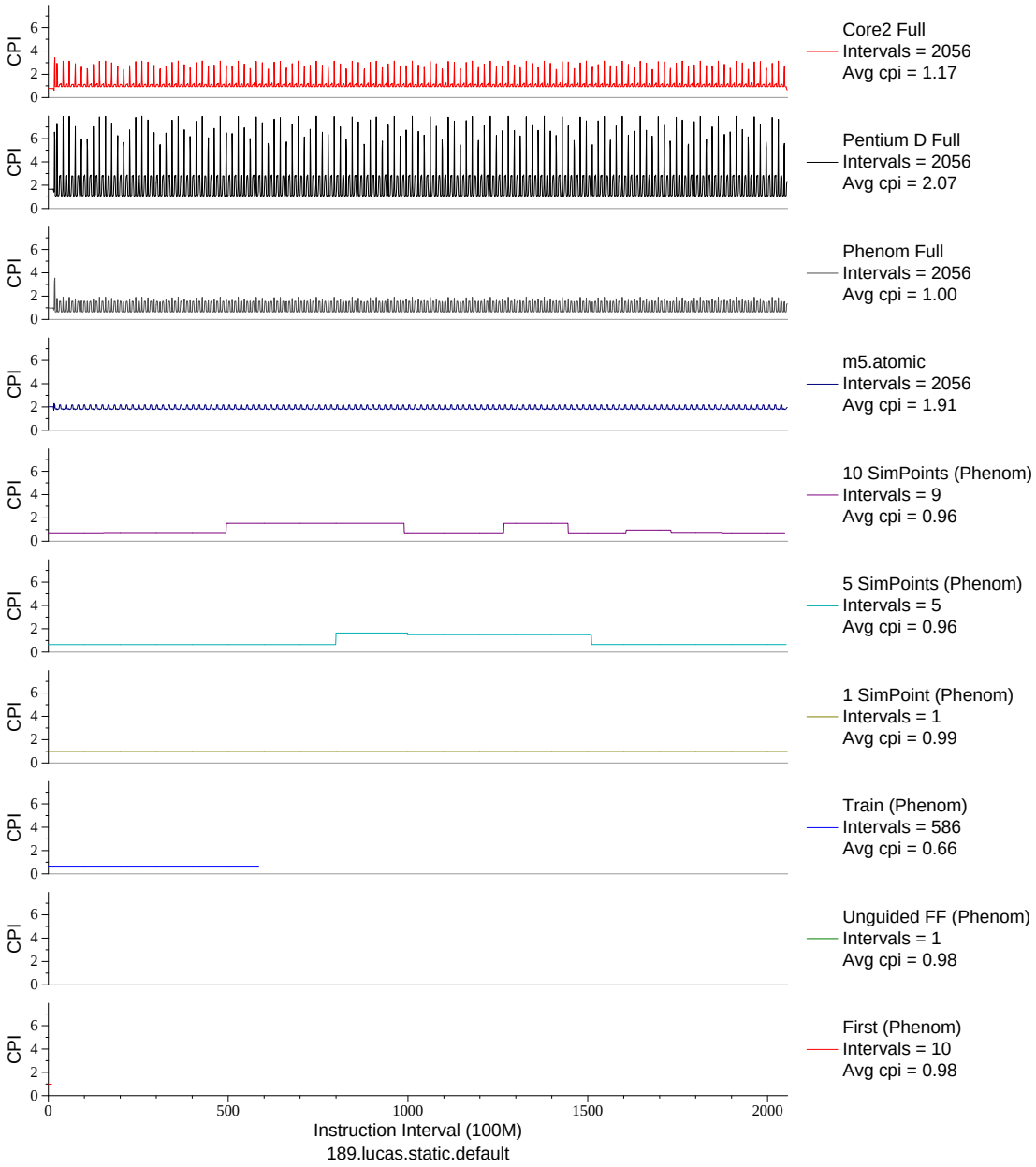

Figure E.74: CPI phase plot for `lucas` (FP, F90, Number Theory)

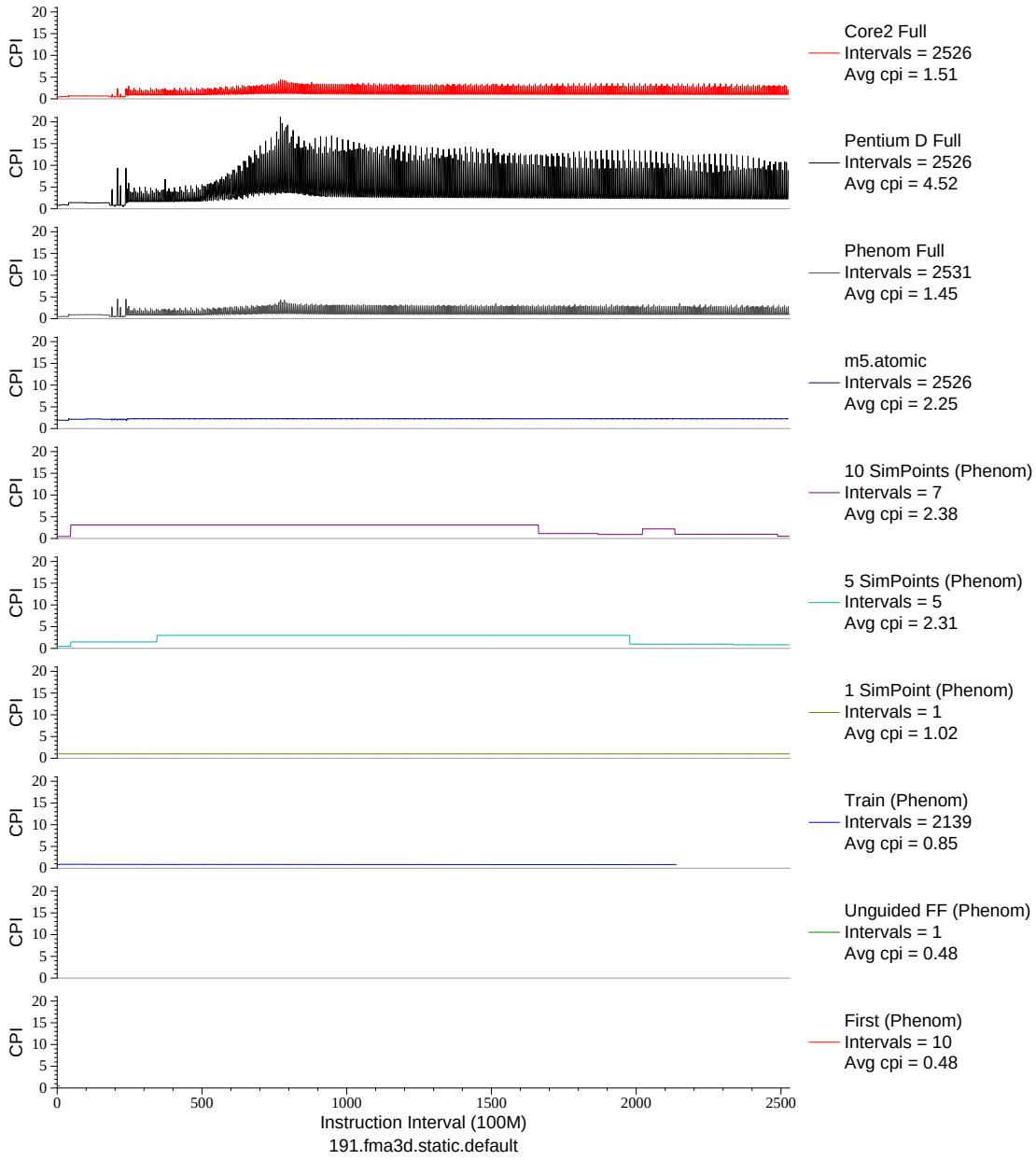

Figure E.75: CPI phase plot for `fma3d` (FP, F90, Crash Simulation)

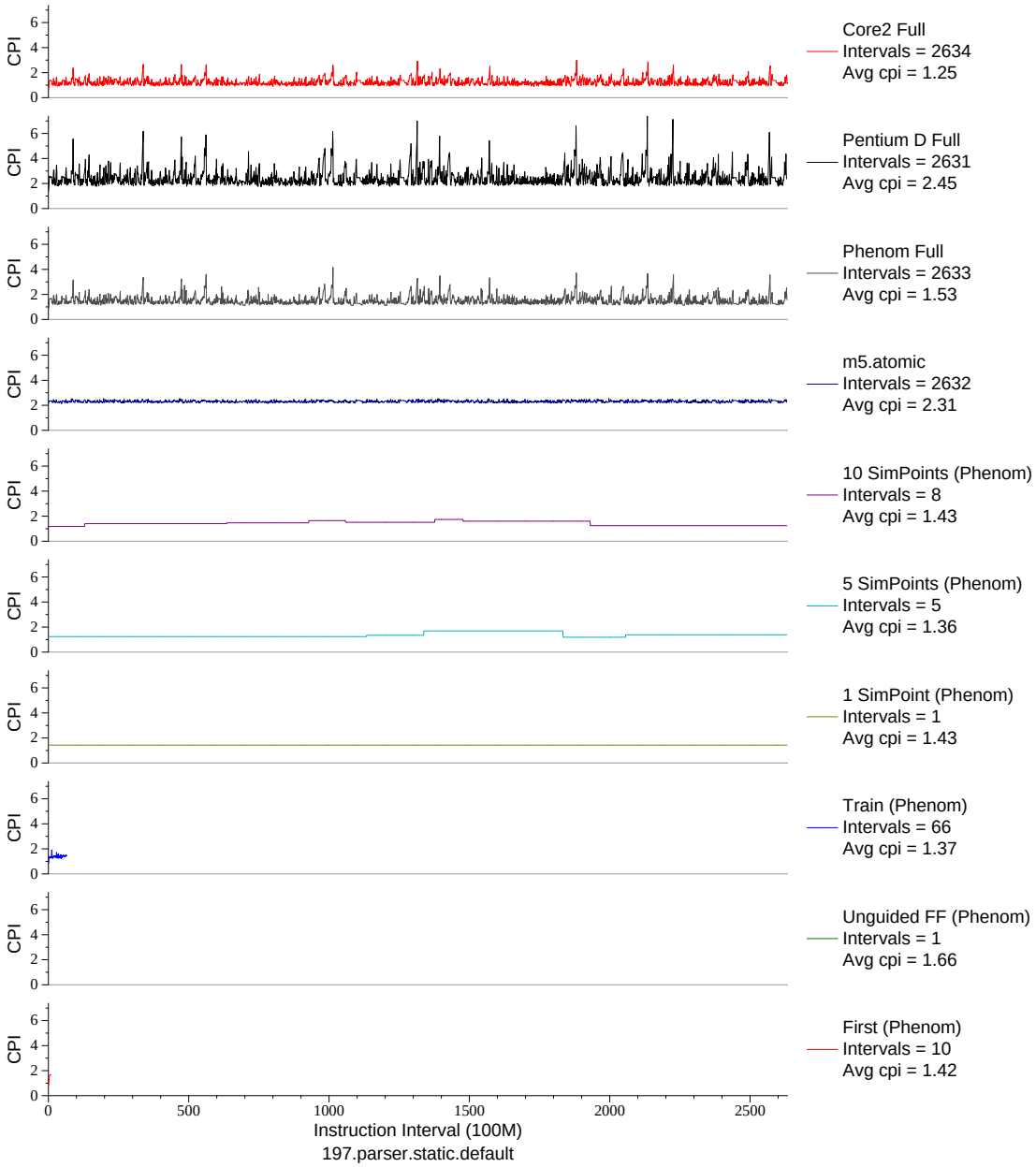

Figure E.76: CPI phase plot for `parser` (INT, C, Word Processing)

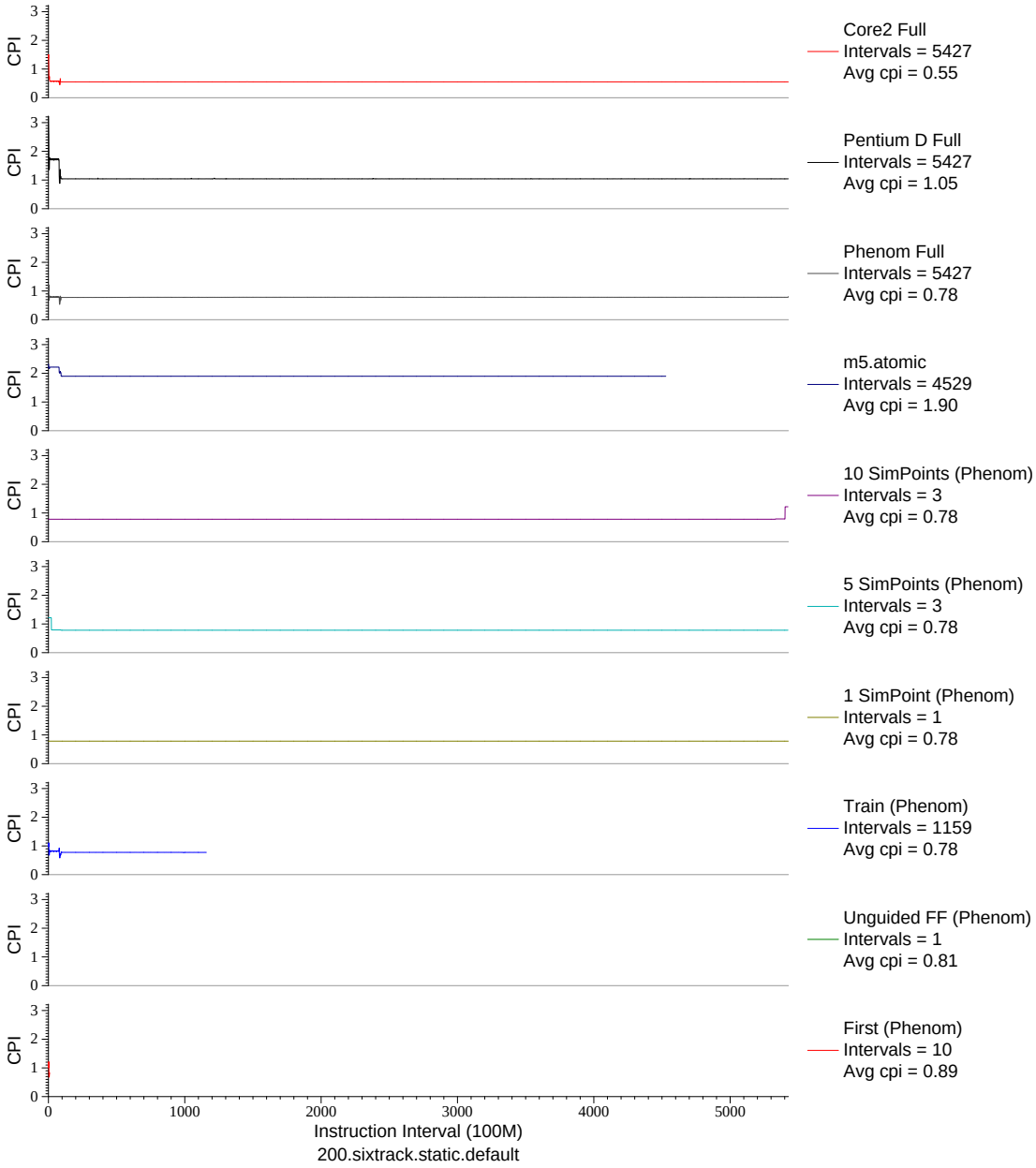

Figure E.77: CPI phase plot for `sixtrack` (FP, F77, Nuclear Physics)

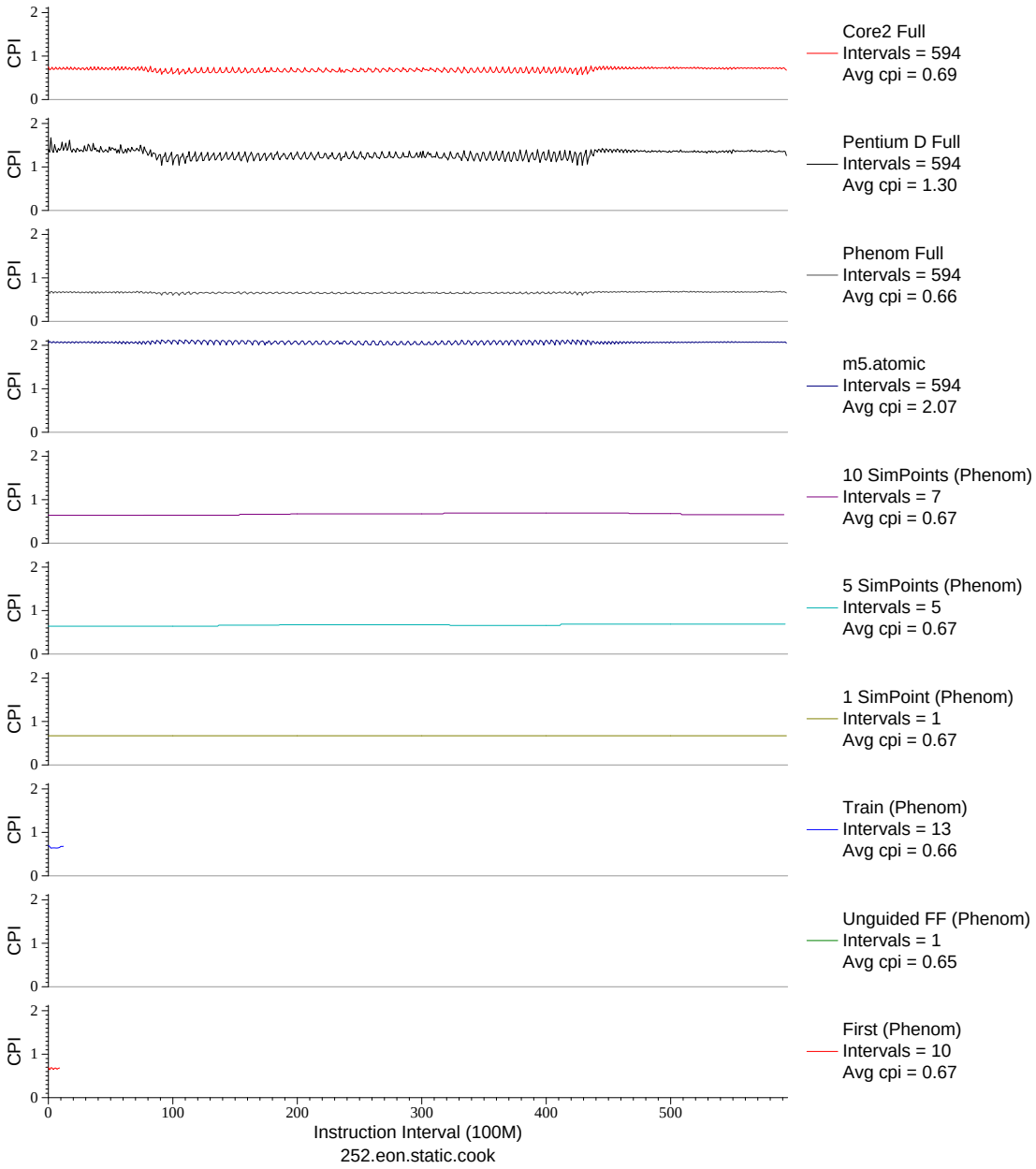

Figure E.78: CPI phase plot for `eon.cook` (INT, C++, Computer Graphics)

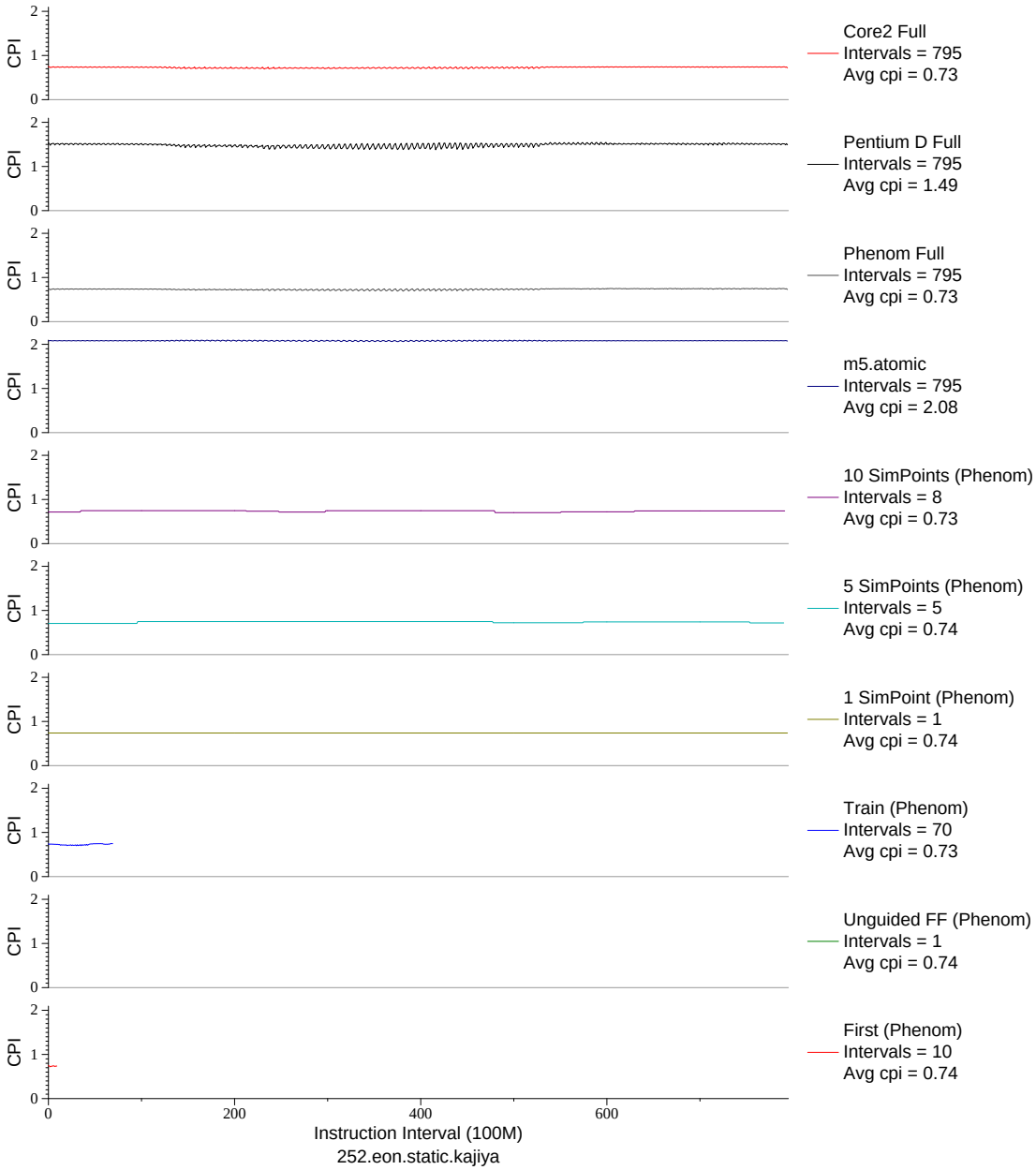

Figure E.79: CPI phase plot for `eon.kaj` (INT, C++, Computer Graphics)

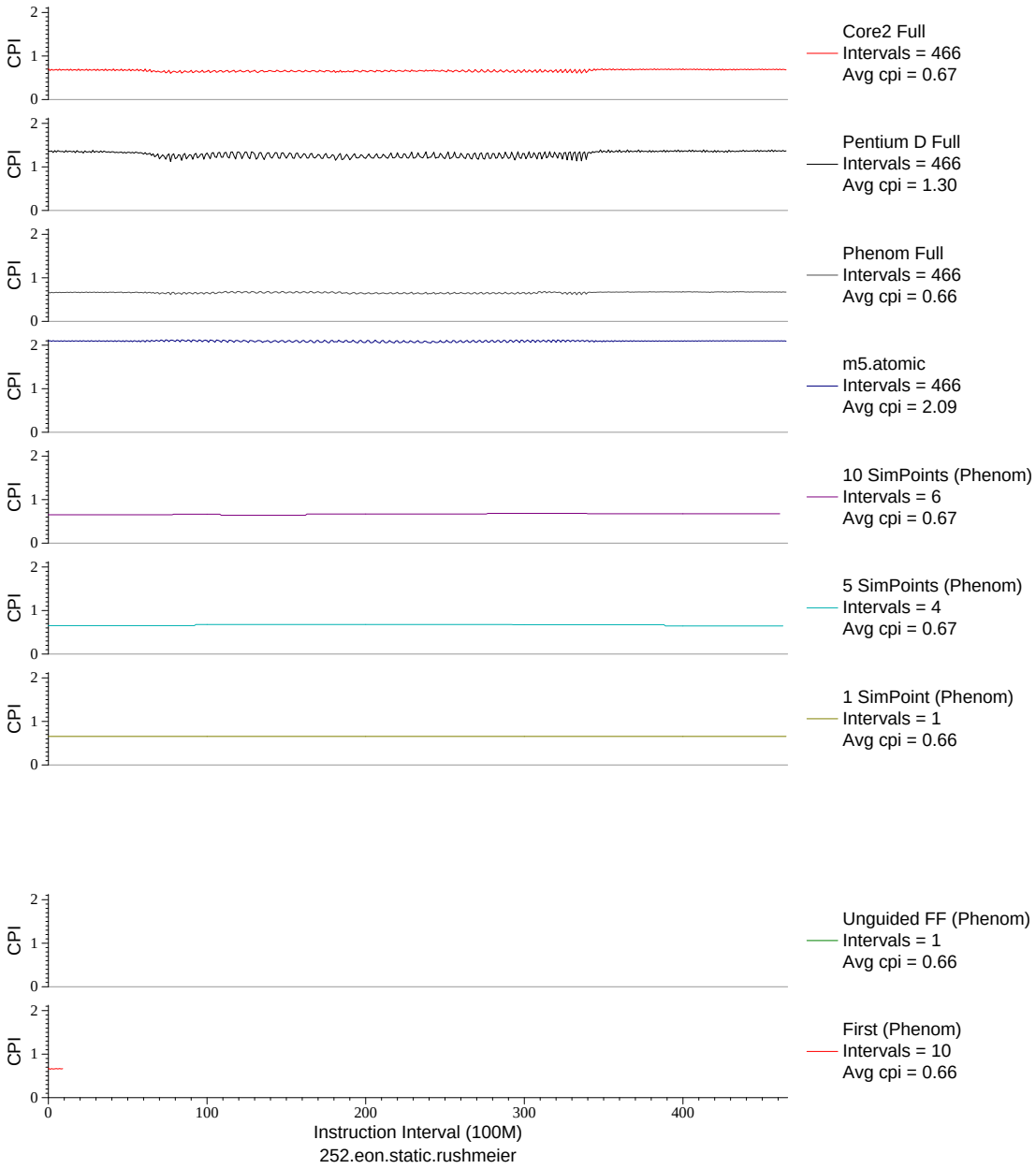

Figure E.80: CPI phase plot for `eon.rush` (INT, C++, Computer Graphics)

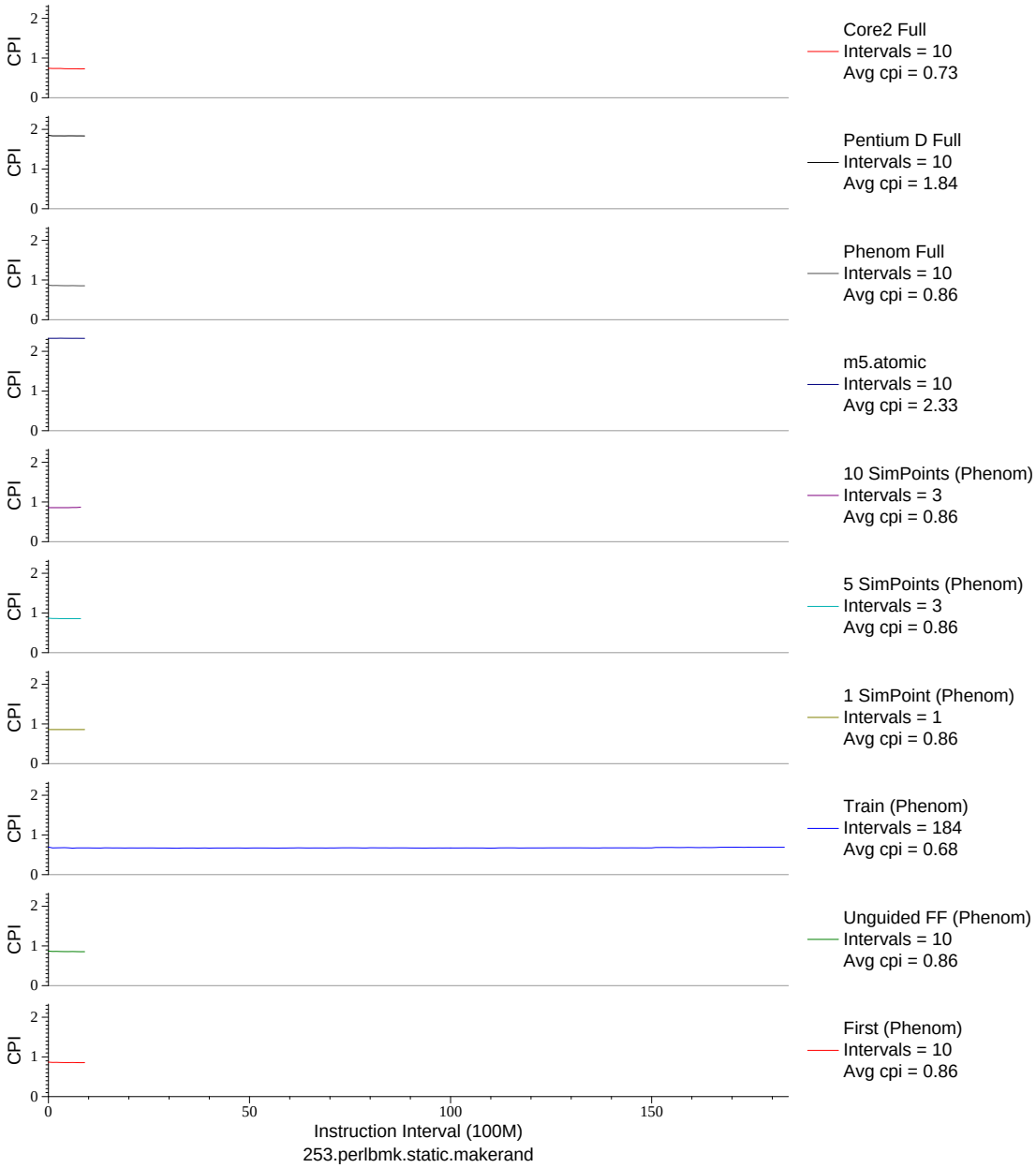

Figure E.81: CPI phase plot for `perlbmk.mkrnd` (INT, C, Scripting Language)

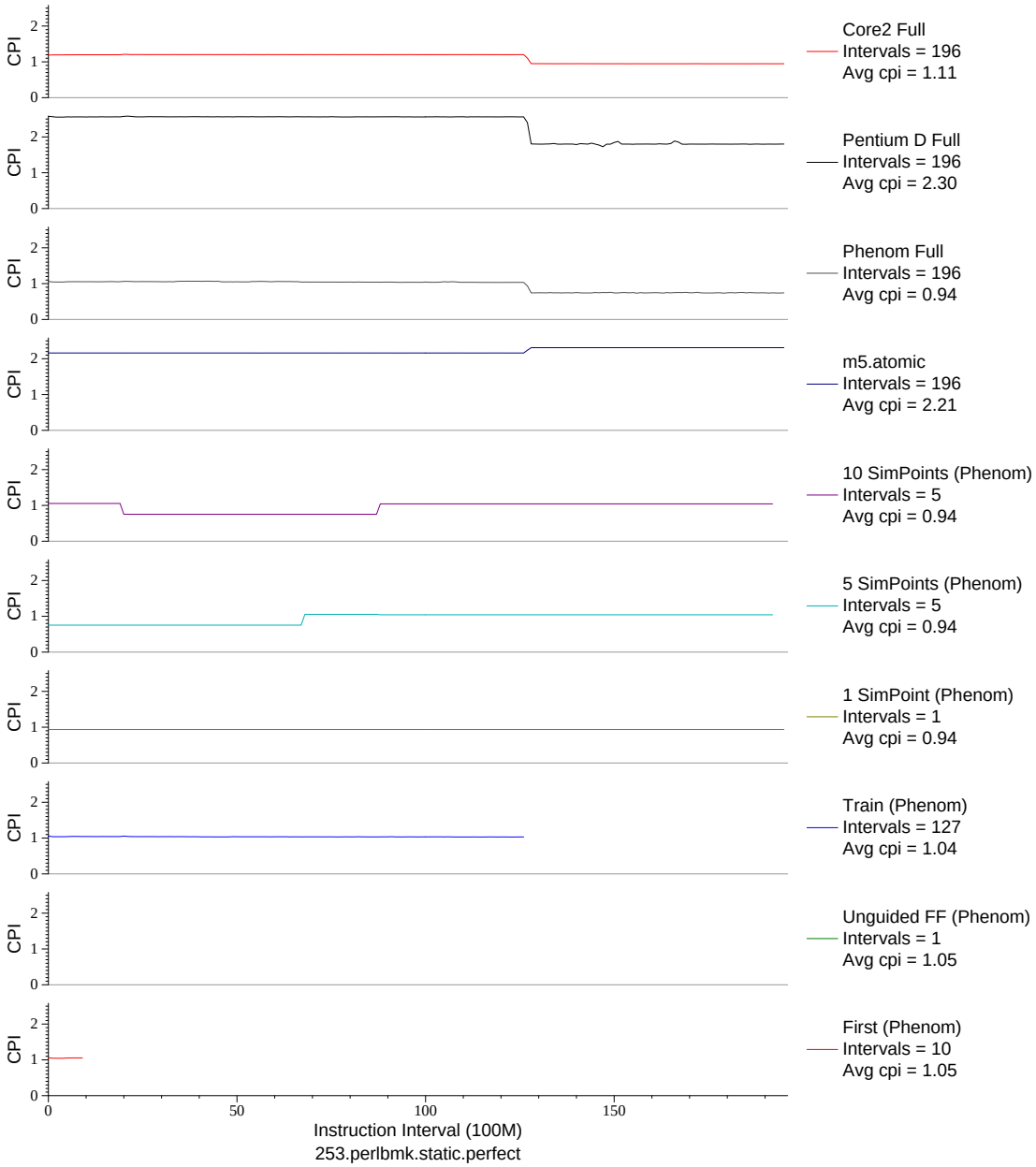

Figure E.82: CPI phase plot for `perlbmk.perf` (INT, C, Scripting Language)

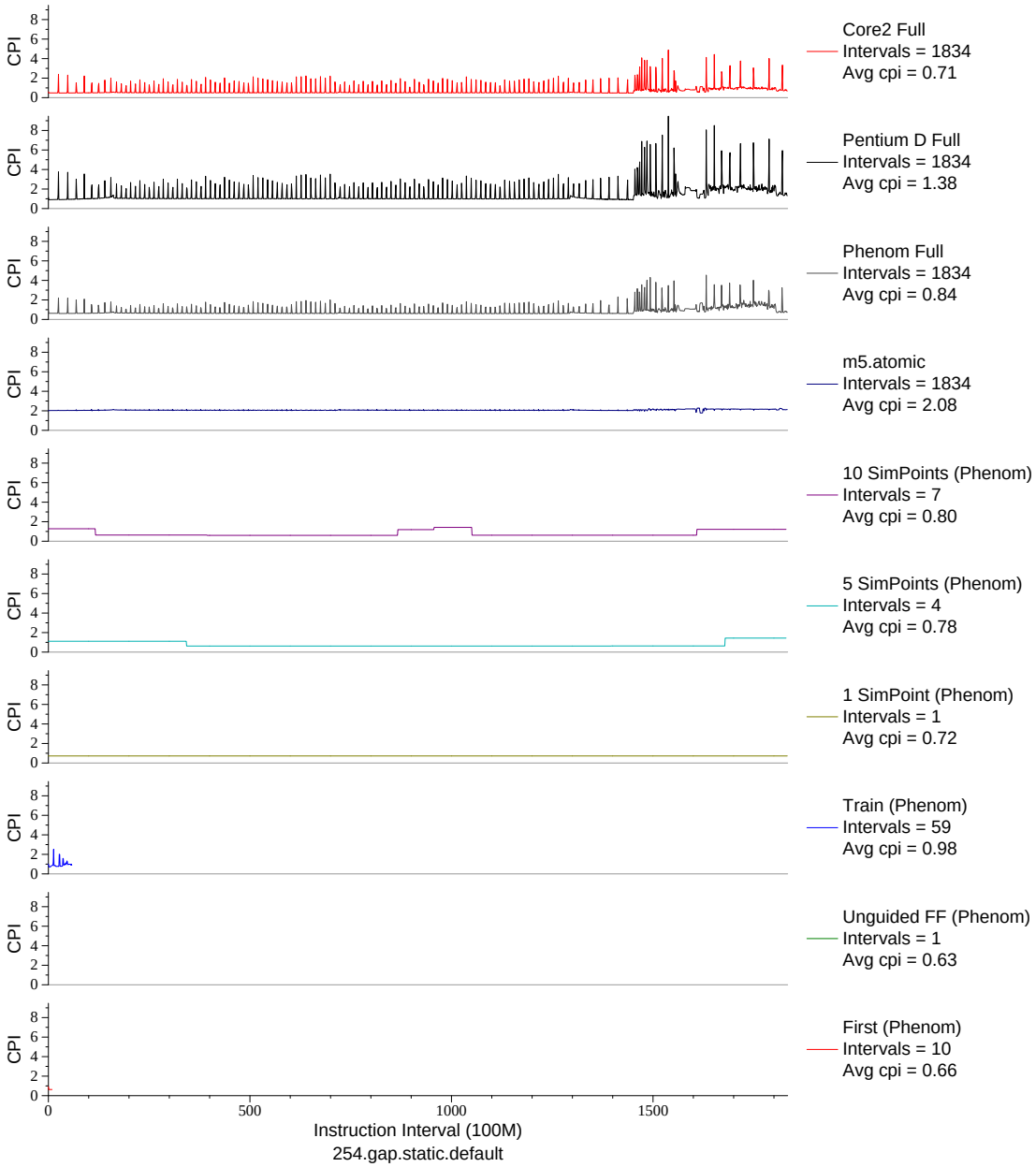

Figure E.83: CPI phase plot for `gap` (INT, C, Group Theory)

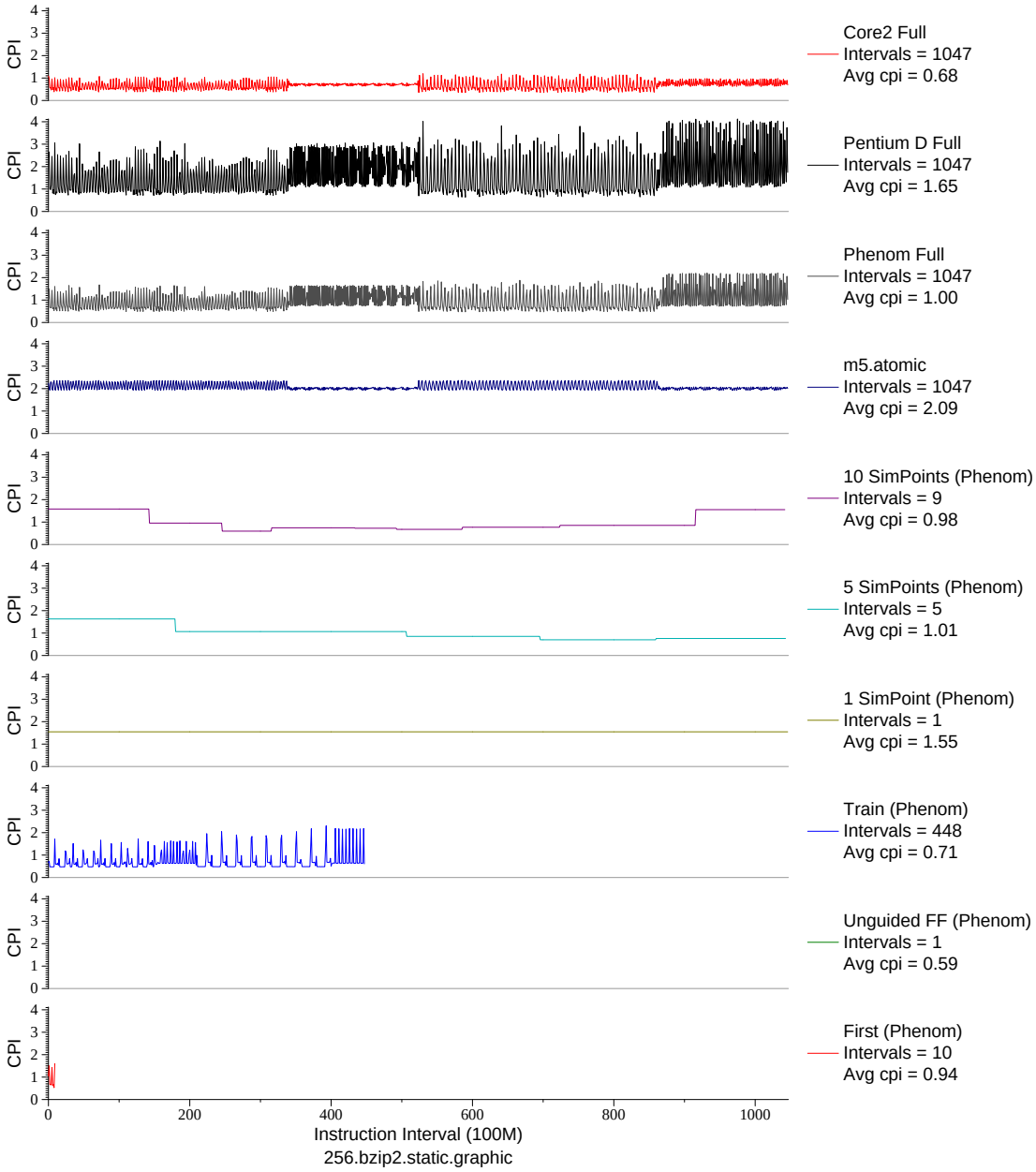

Figure E.84: CPI phase plot for `bzip2.graph` (INT, C, Compression)

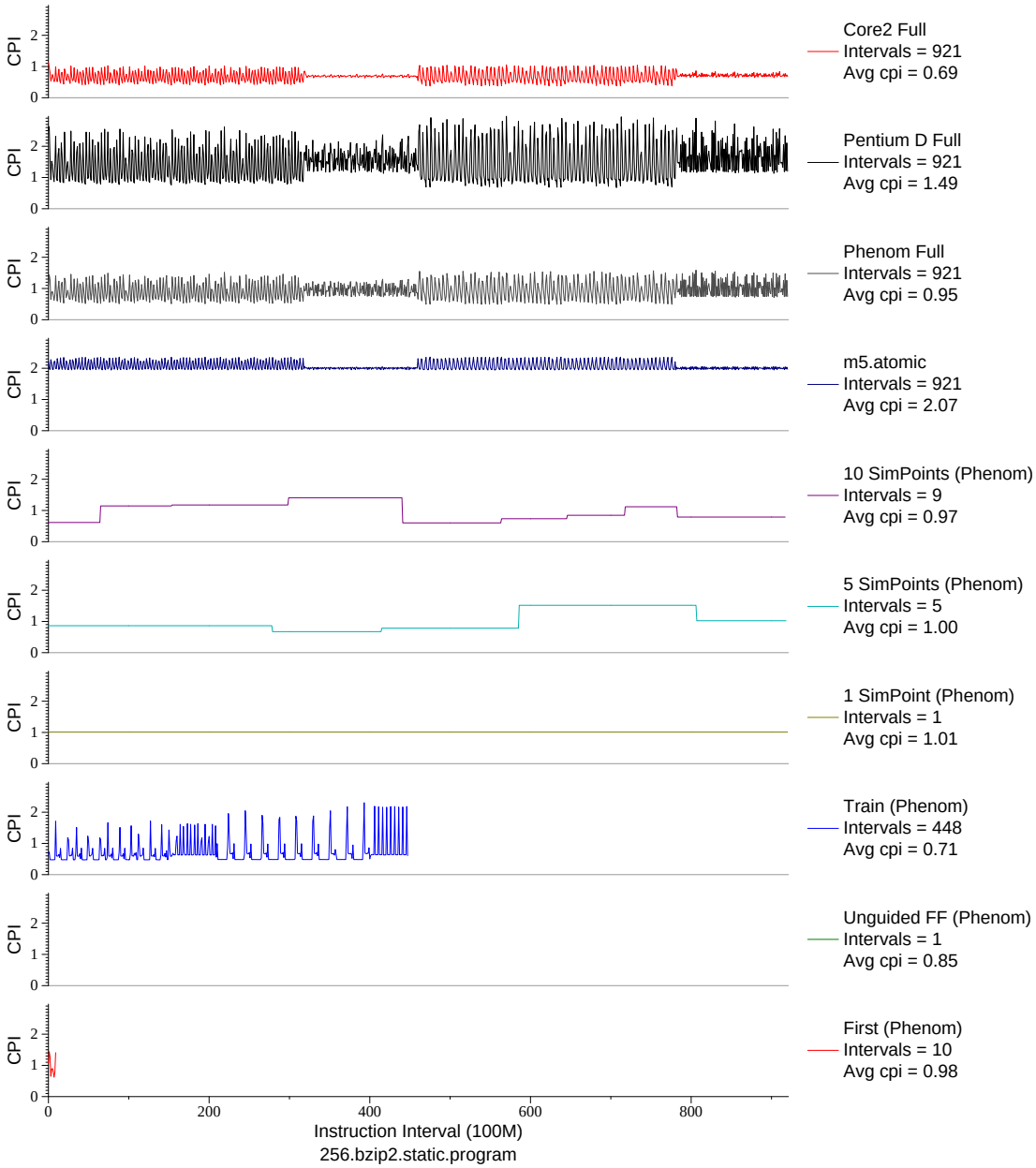

Figure E.85: CPI phase plot for `bzip2.prog` (INT, C, Compression)

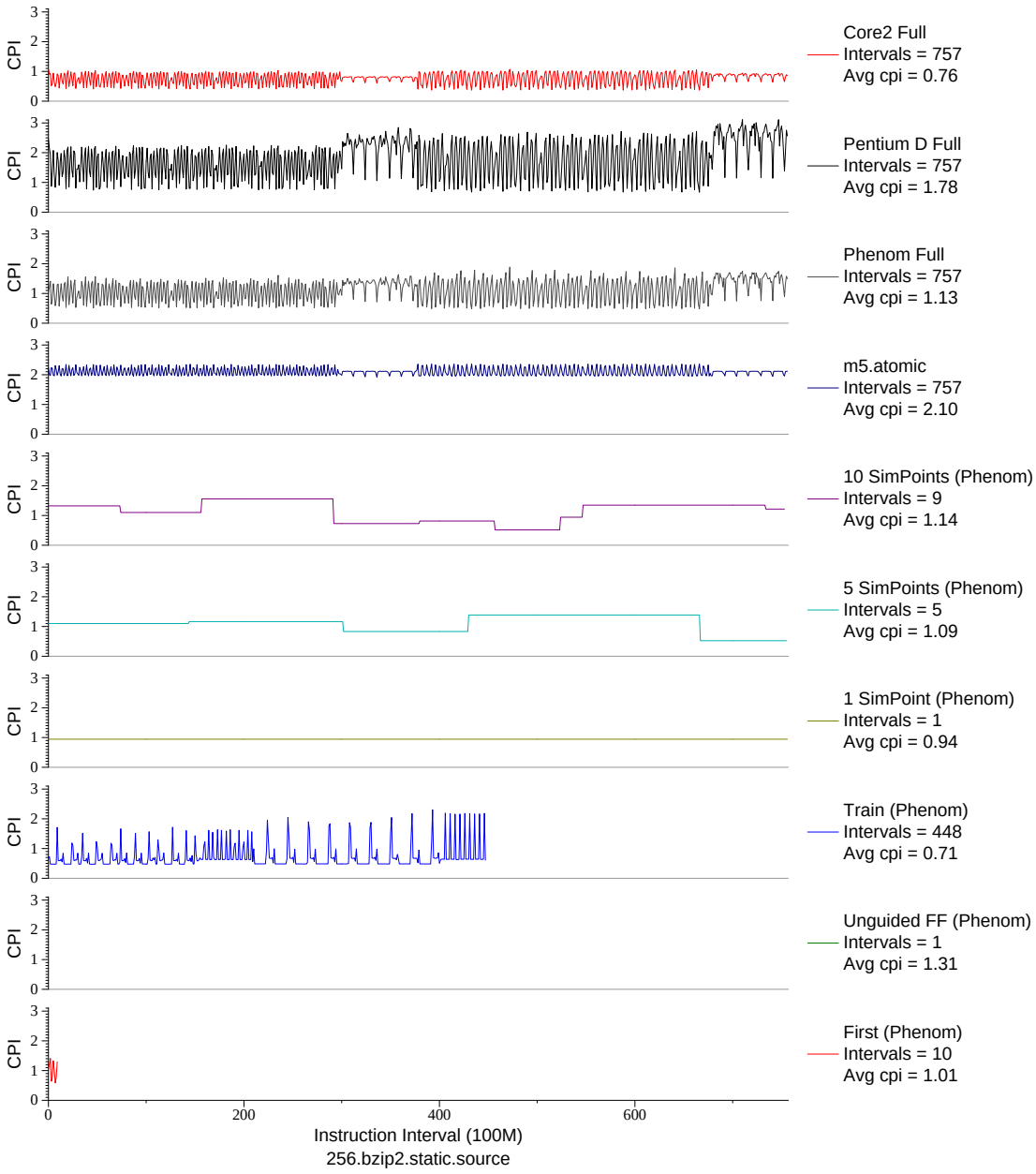

Figure E.86: CPI phase plot for `bzip2.src` (INT, C, Compression)

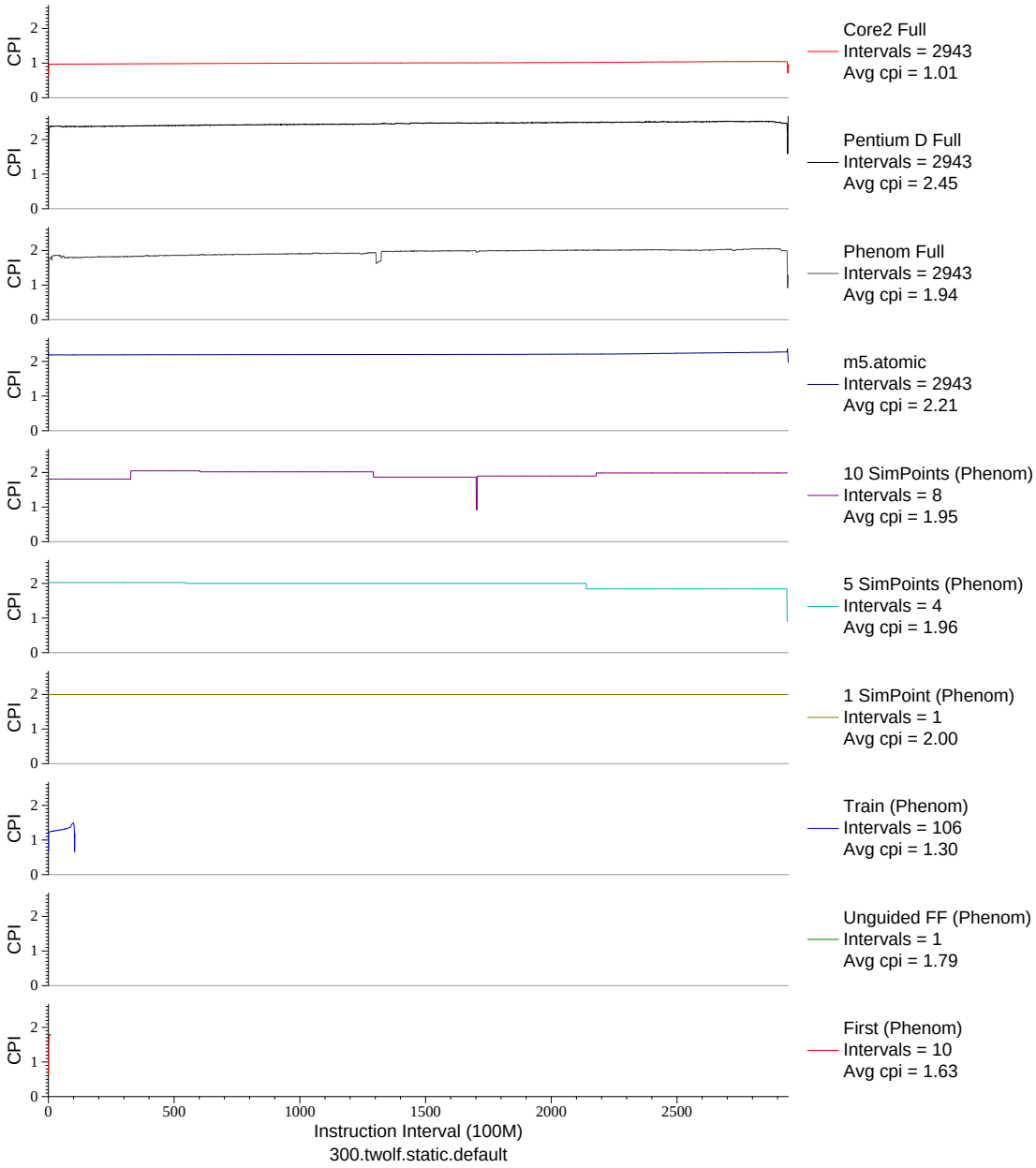

Figure E.87: CPI phase plot for `twolf` (INT, C, Place/Route)

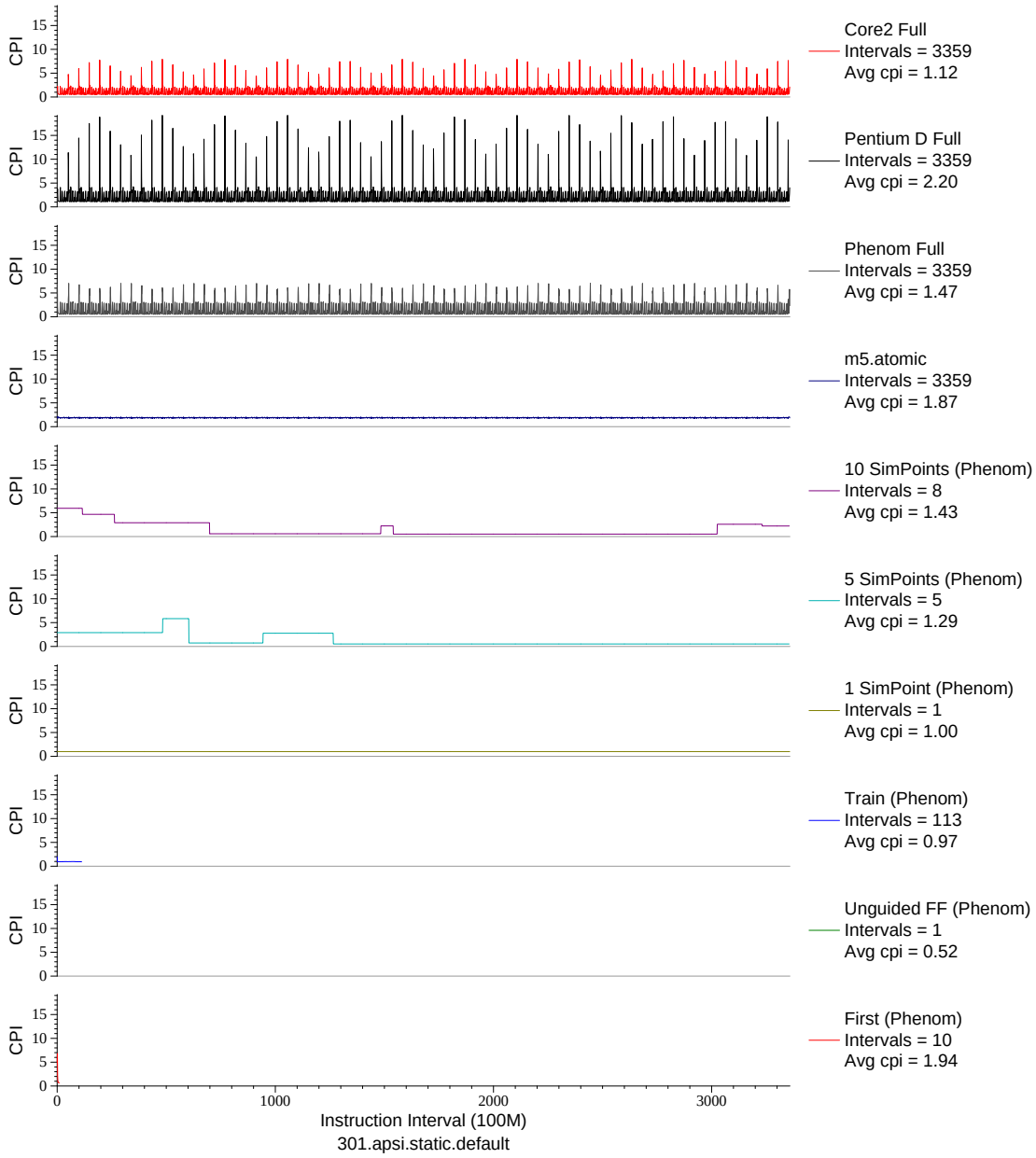

Figure E.88: CPI phase plot for `apsi` (FP, F77, Meteorology/Pollution)

APPENDIX F

MULTI-ARCHITECTURE PHASE PLOTS

Program phases are often similar across architectures; the overall work being done is the same even though the various platforms involved have different underlying microarchitecture. Despite the gross similarities, the cycle and retired instruction counts differ enough that SimPoint intervals gathered for one architecture cannot be used to analyze executions on another. Below are phase plots for the cycles per instruction (CPI) metric for the SPEC CPU2000 benchmarks, running on the MIPS, ia64, x86, x86_64 platforms. Some of the plots are missing; this is due to the performance counter results being unavailable for that particular benchmark and architecture combination.

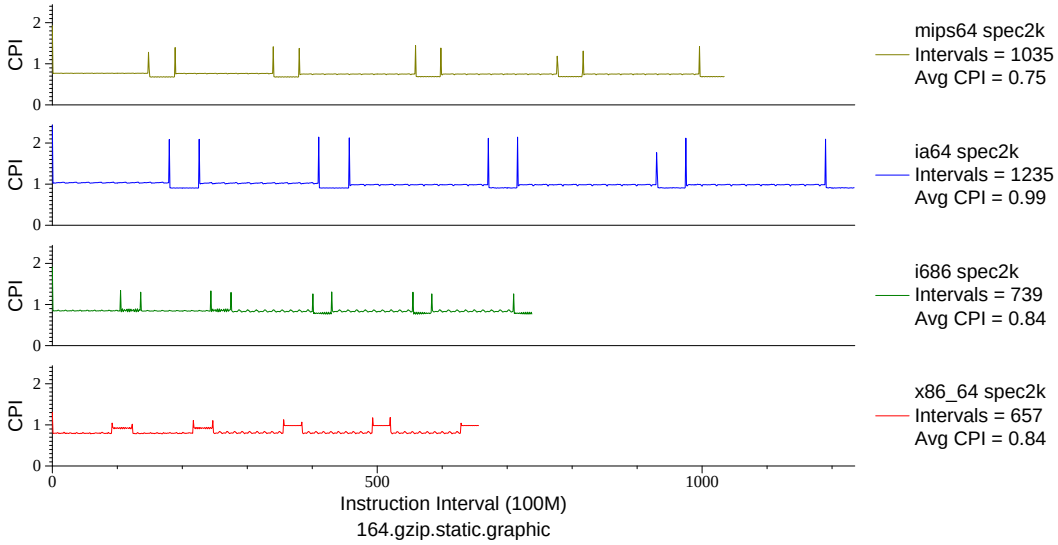

Figure F.1: Multi-arch CPI plot for `gzip.graph` (INT, C, Compression)

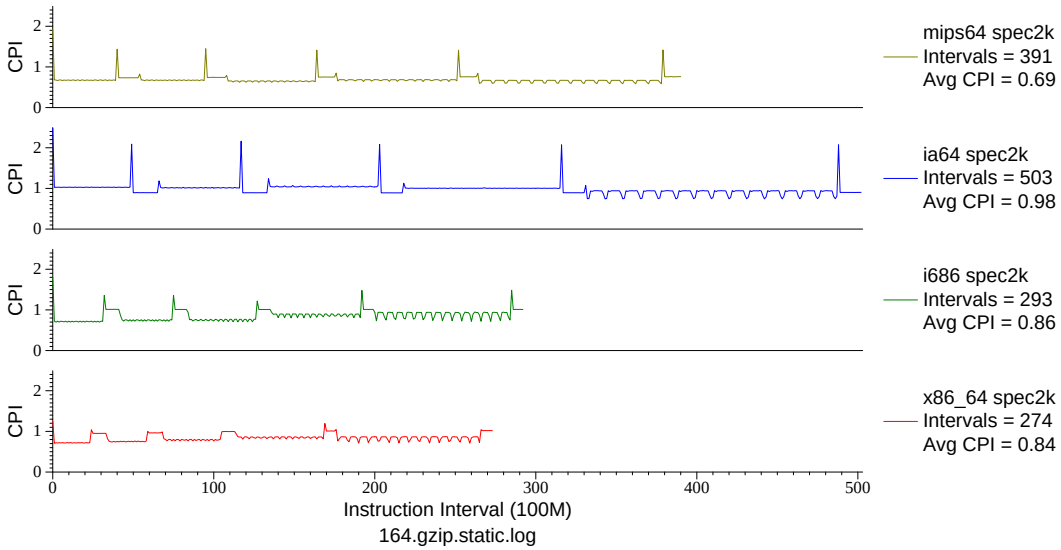

Figure F.2: Multi-arch CPI plot for `gzip.log` (INT, C, Compression)

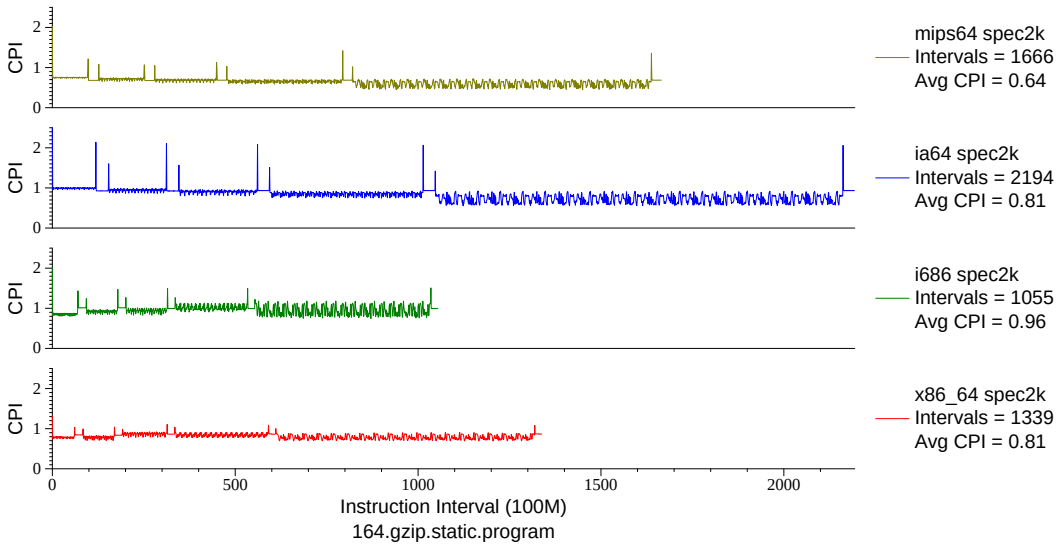

Figure F.3: Multi-arch CPI plot for `gzip.prog` (INT, C, Compression)

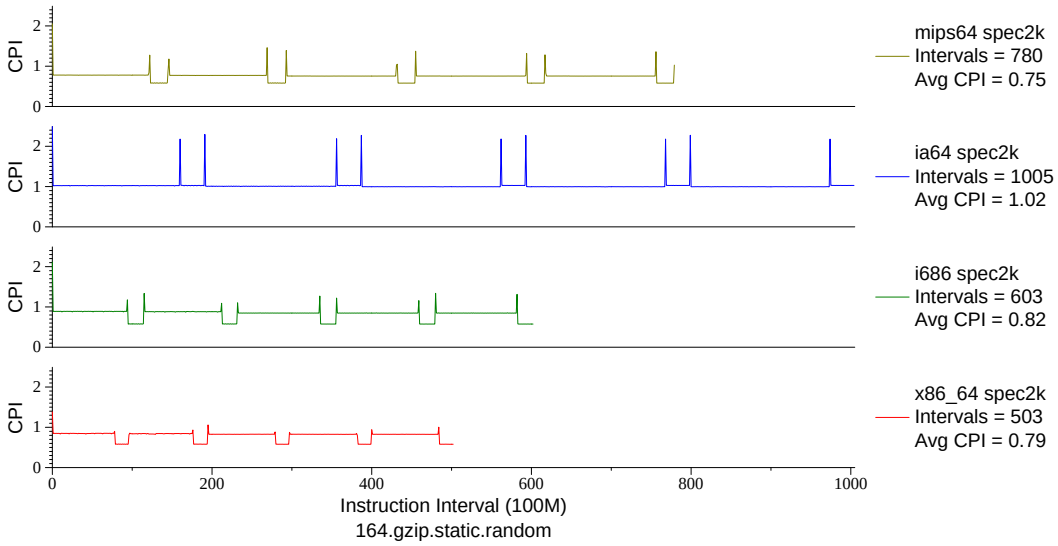

Figure F.4: Multi-arch CPI plot for `gzip.rand` (INT, C, Compression)

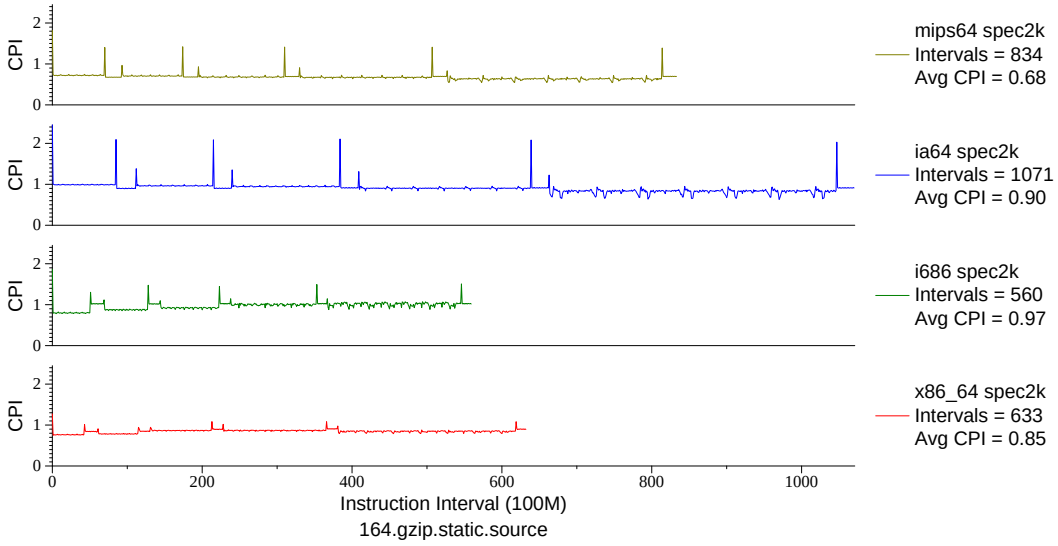

Figure F.5: Multi-arch CPI plot for `gzip.src` (INT, C, Compression)

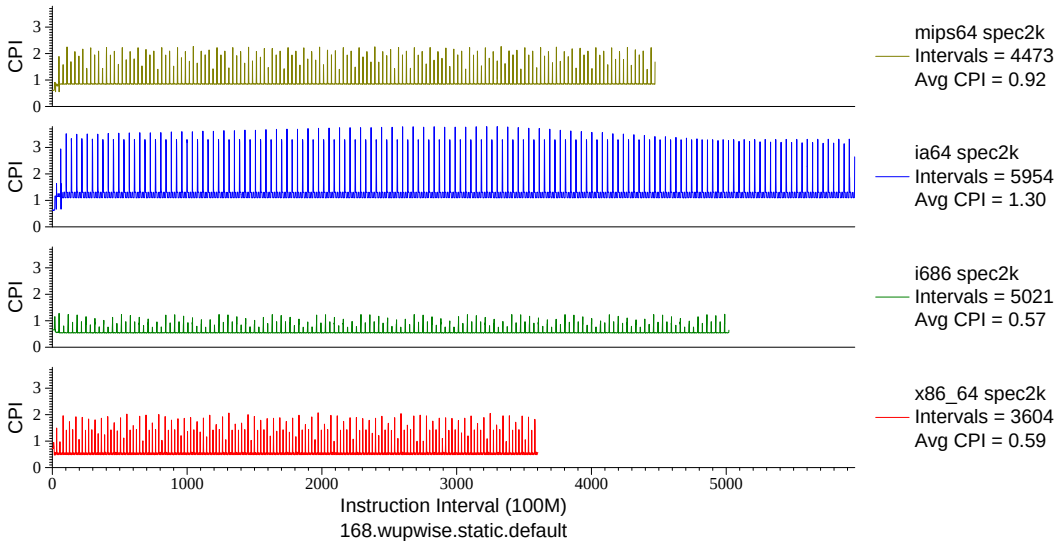

Figure F.6: Multi-arch CPI plot for `wupwise` (FP, F77, Quantum Chromodynamics)

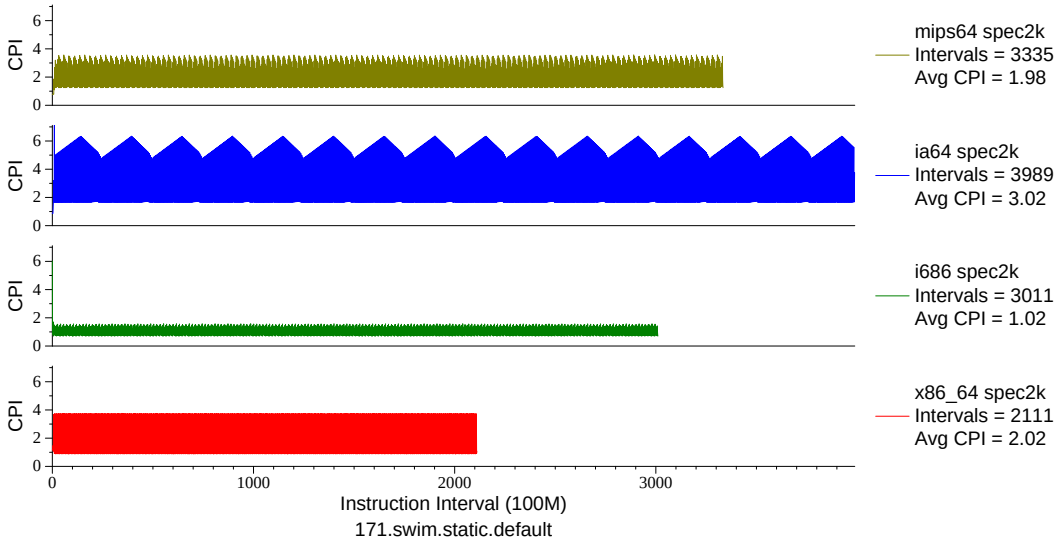

Figure F.7: Multi-arch CPI plot for `swim` (FP, F77, Meteorology/Water)

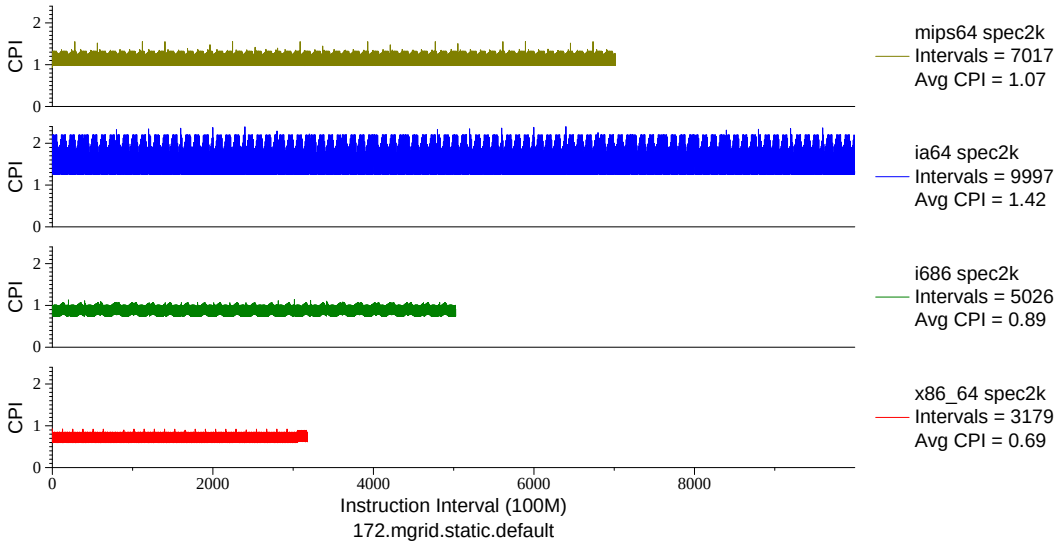

Figure F.8: Multi-arch CPI plot for `mgrid` (FP, F77, Multi-Grid Solver)

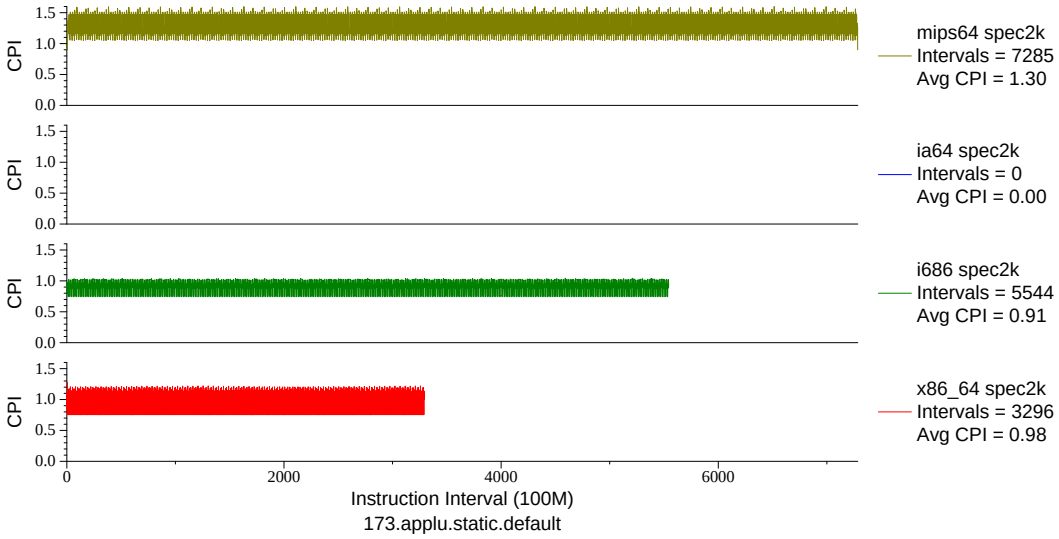

Figure F.9: Multi-arch CPI plot for `applu` (FP, F77, Fluid Dynamics)

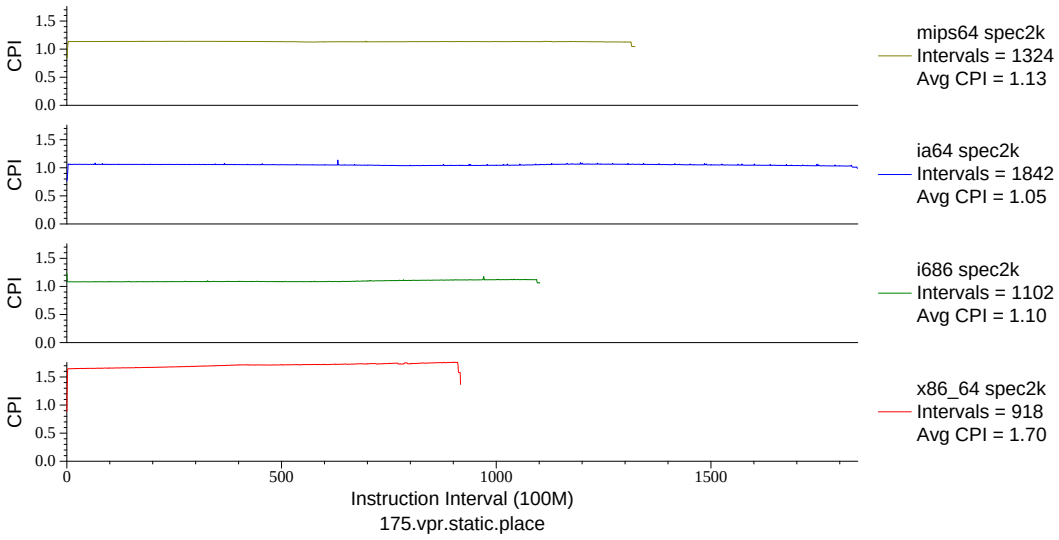

Figure F.10: Multi-arch CPI plot for `vpr.place` (INT, C, FPGA Place/Route)

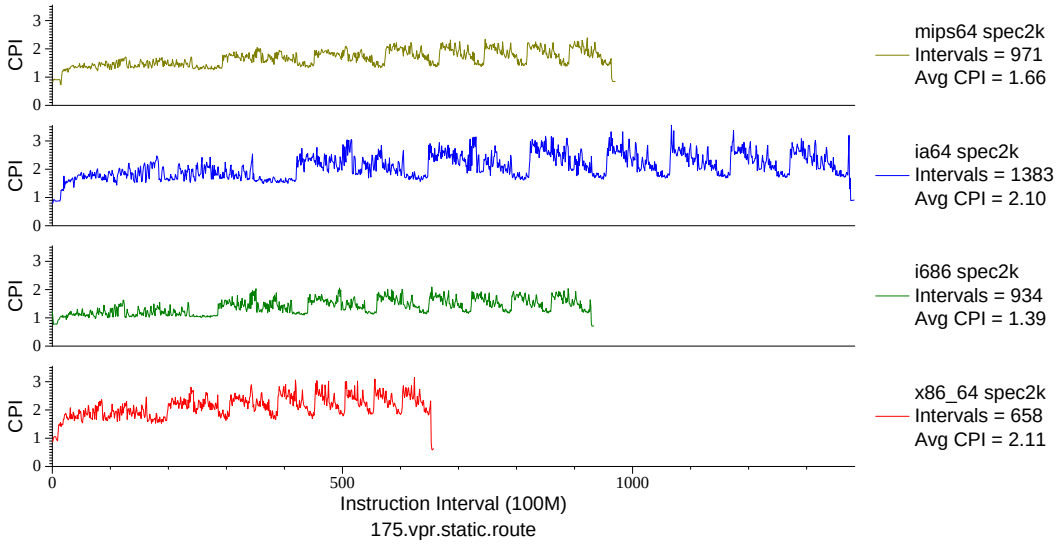

Figure F.11: Multi-arch CPI plot for `vpr.route` (INT, C, FPGA Place/Route)

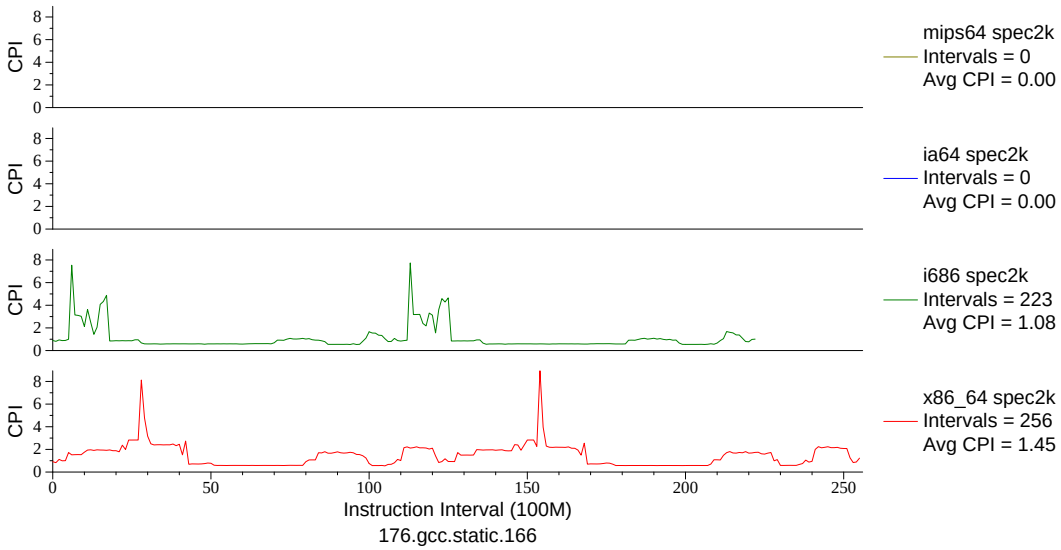

Figure F.12: Multi-arch CPI plot for `gcc.166` (INT, C, C Compiler)

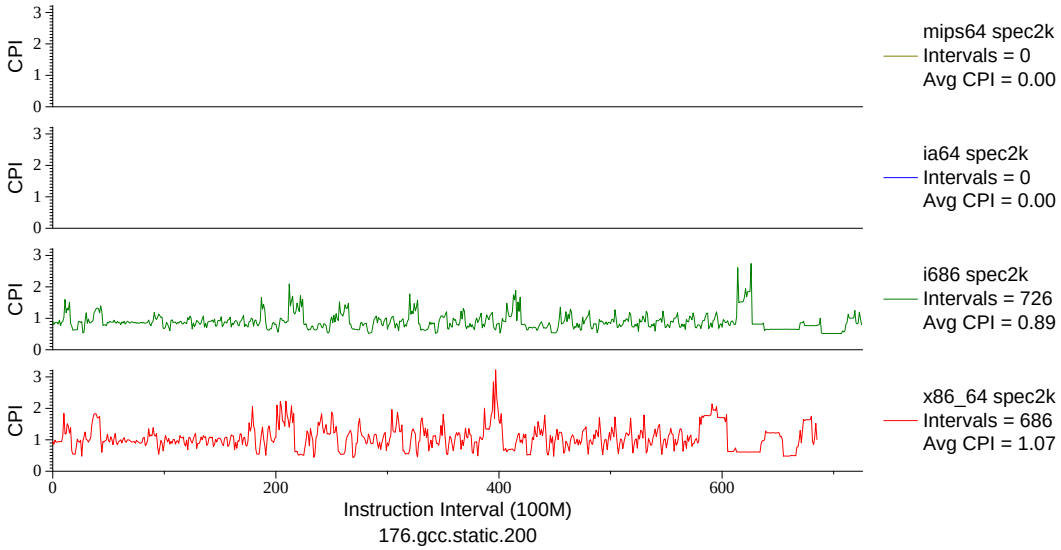

Figure F.13: Multi-arch CPI plot for `gcc.200` (INT, C, C Compiler)

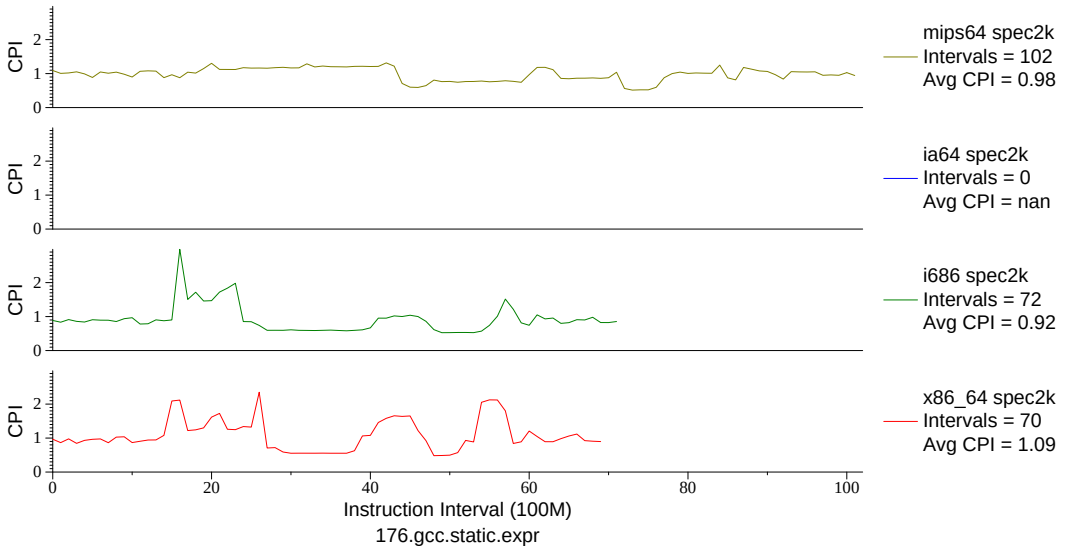

Figure F.14: Multi-arch CPI plot for `gcc.expr` (INT, C, C Compiler)

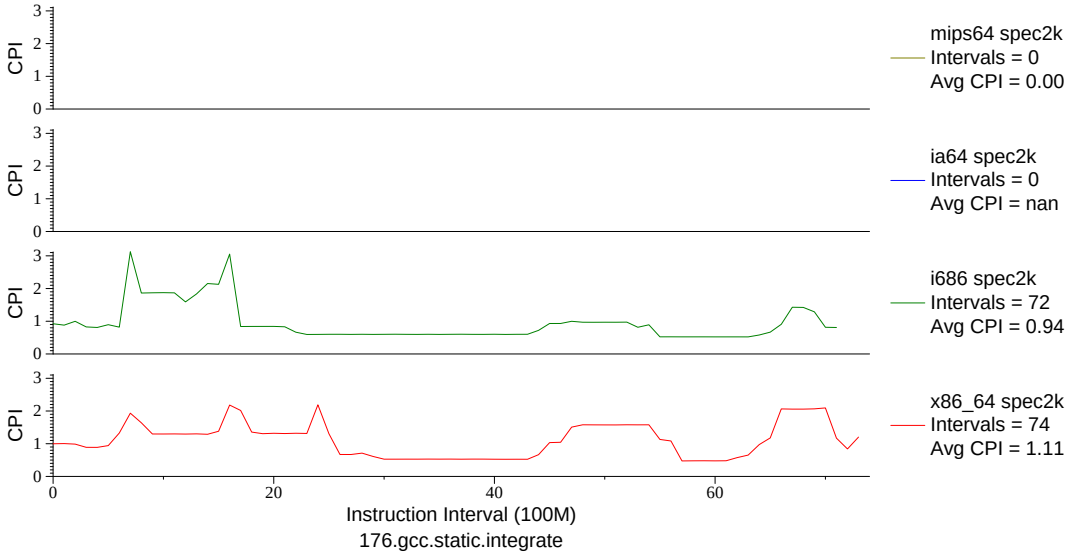

Figure F.15: Multi-arch CPI plot for `gcc.integrate` (INT, C, C Compiler)

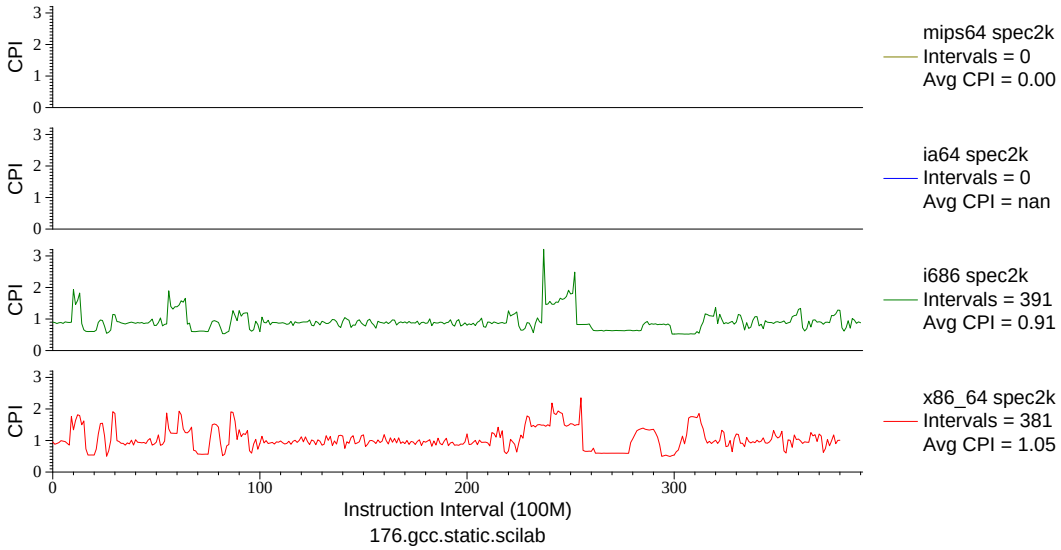

Figure F.16: Multi-arch CPI plot for `gcc.scilab` (INT, C, C Compiler)

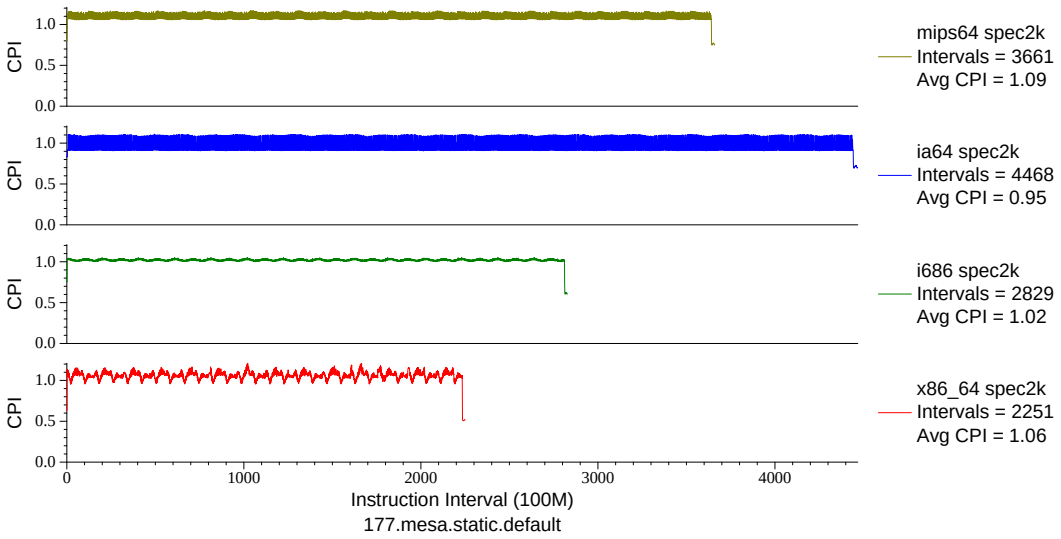

Figure F.17: Multi-arch CPI plot for `mesa` (FP, C, 3D-graphics)

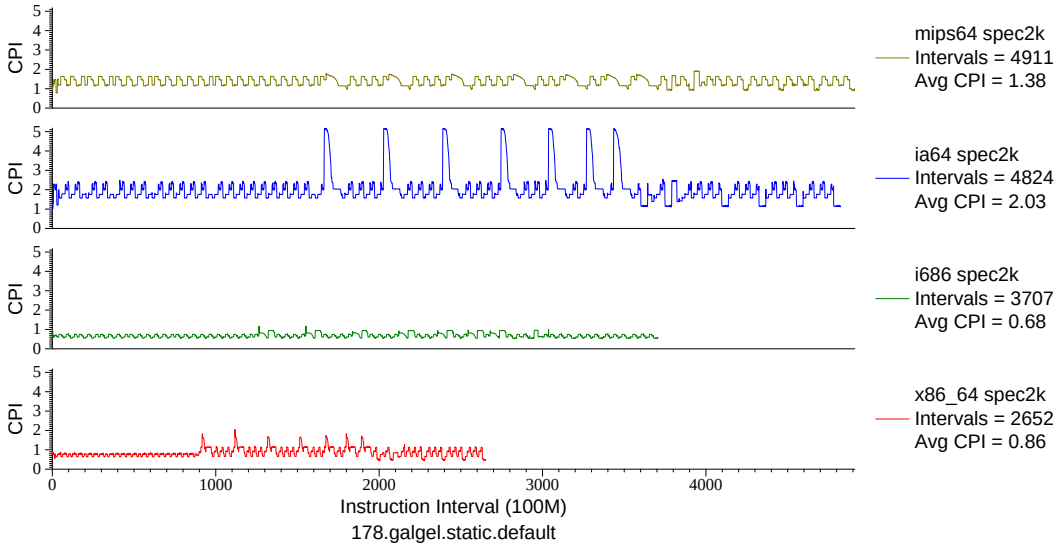

Figure F.18: Multi-arch CPI plot for `galgel` (FP, F90, Fluid Dynamics)

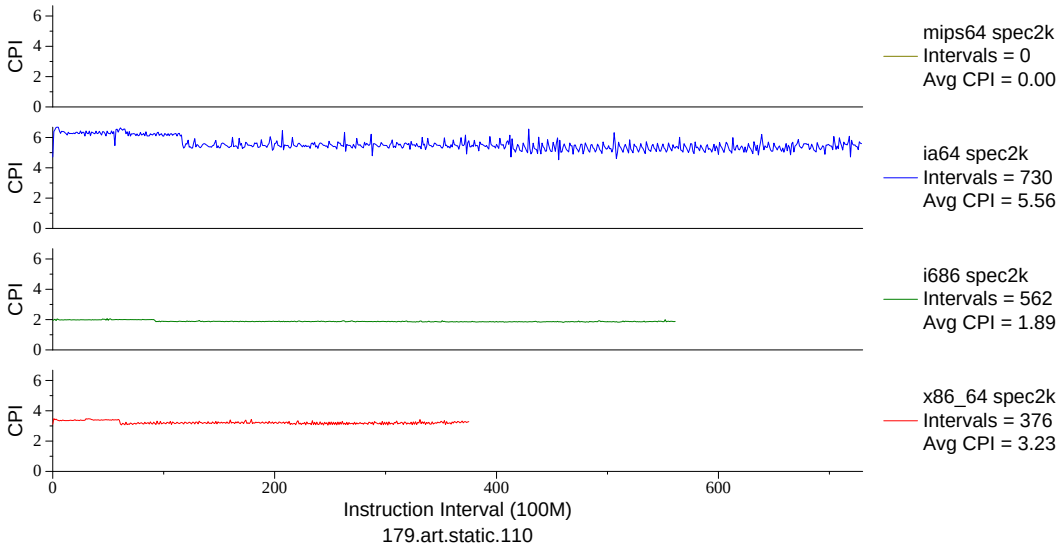

Figure F.19: Multi-arch CPI plot for `art.110` (FP, C, Neural Networks)

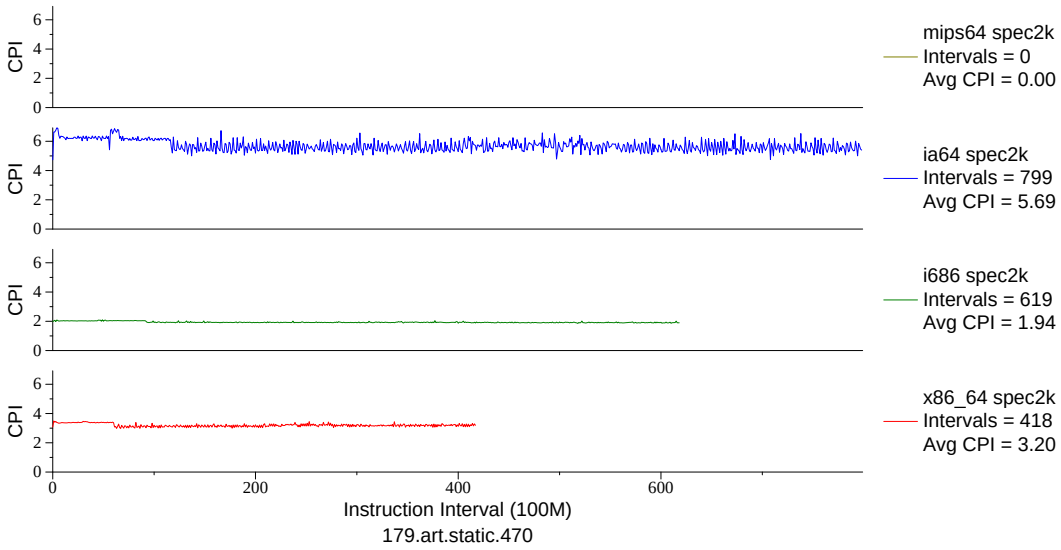

Figure F.20: Multi-arch CPI plot for `art.470` (FP, C, Neural Networks)

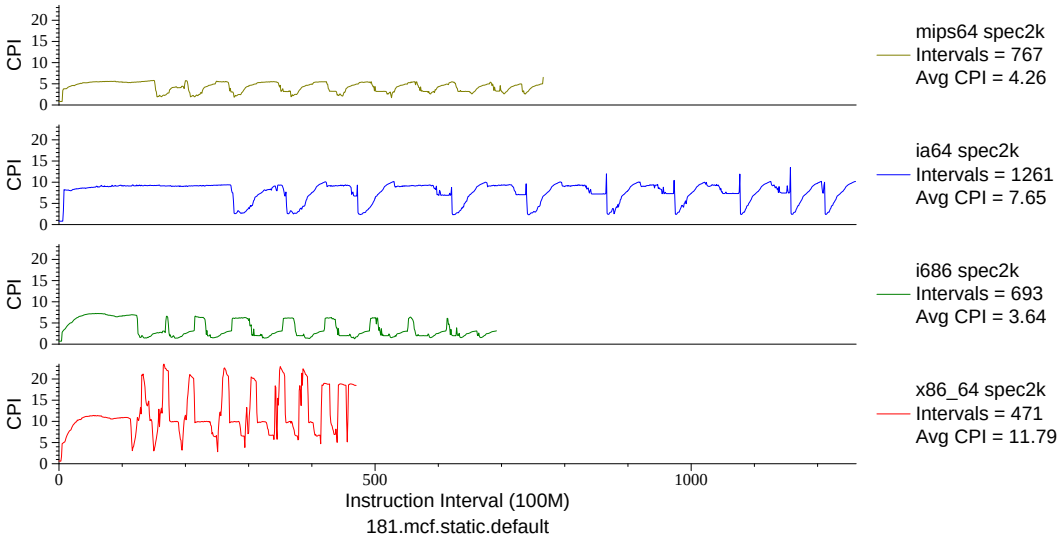

Figure F.21: Multi-arch CPI plot for `mcf` (INT, C, Combinatorial Opt)

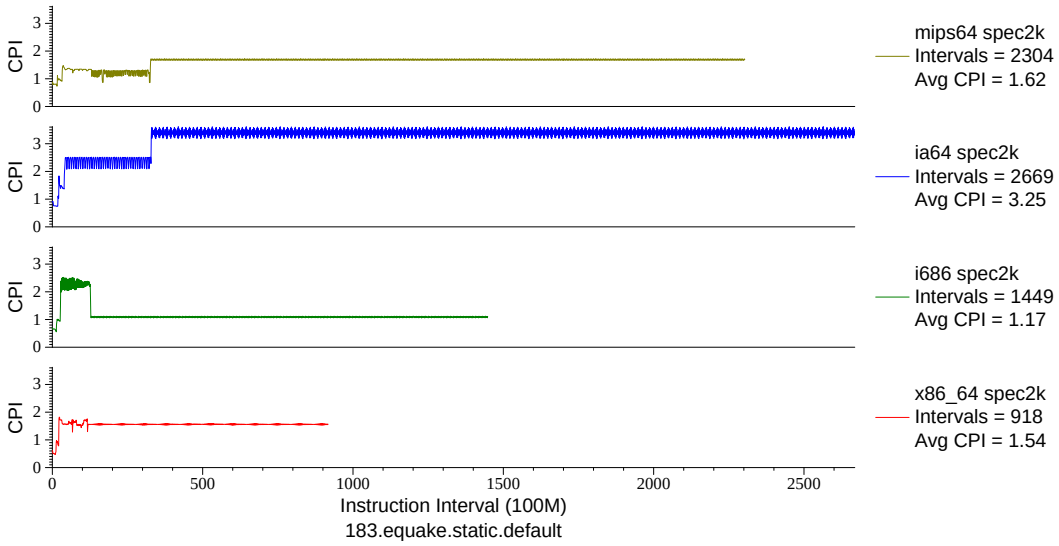

Figure F.22: Multi-arch CPI plot for `equake` (FP, C, Seismic Propogation)

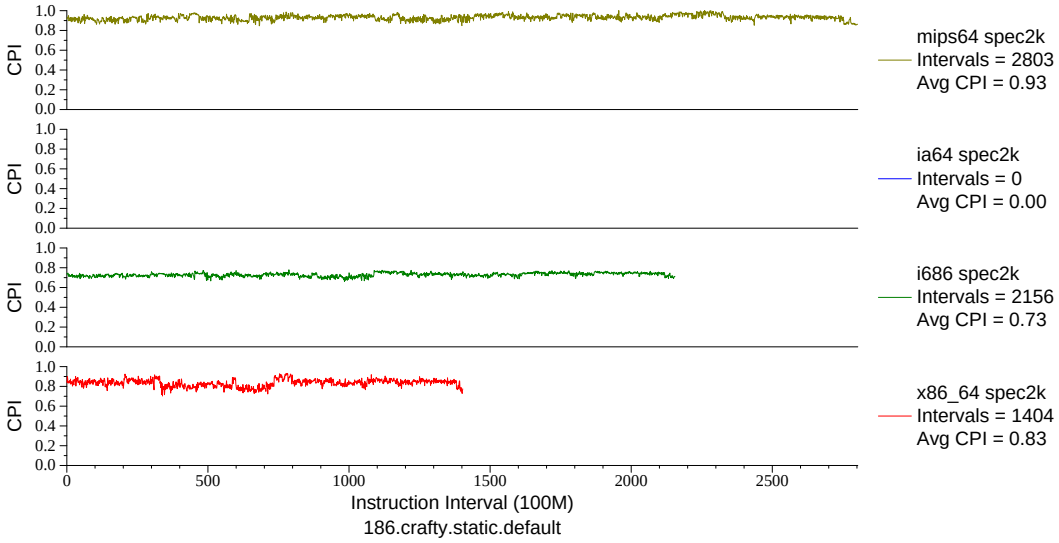

Figure F.23: Multi-arch CPI plot for `crafty` (INT, C, Chess)

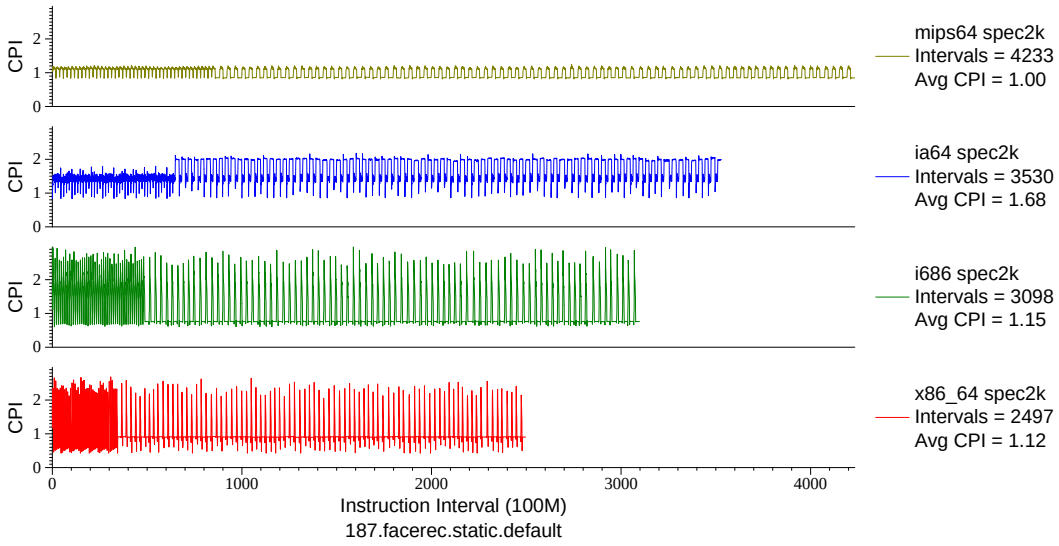

Figure F.24: Multi-arch CPI plot for `facerec` (FP, F90, Facial Recognition)

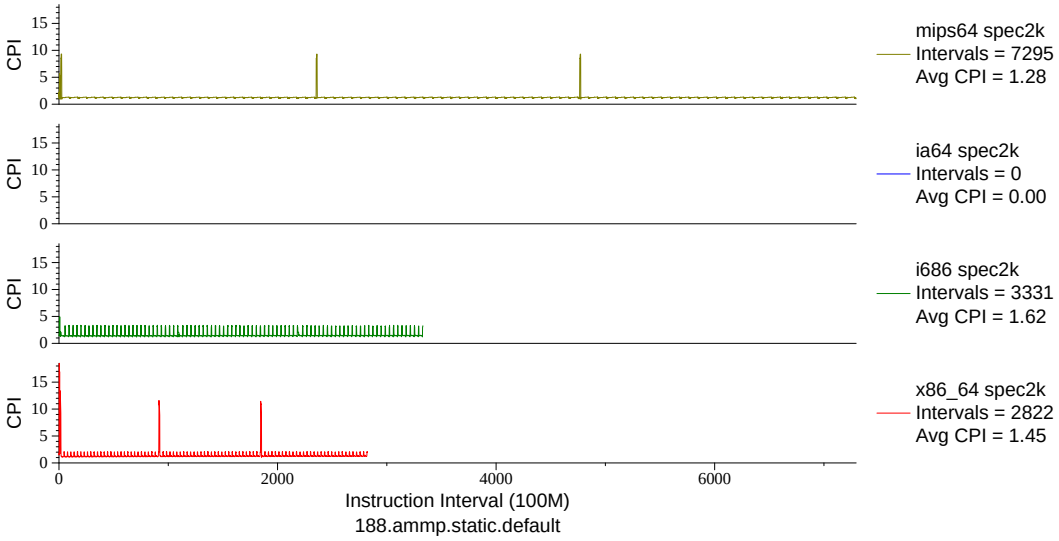

Figure F.25: Multi-arch CPI plot for `ammp` (FP, C, Chemistry)

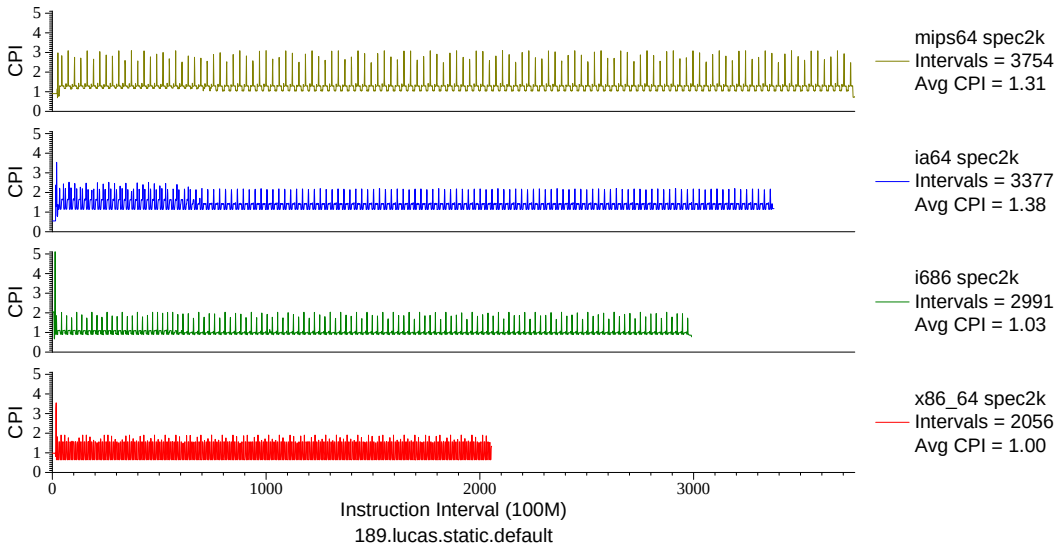

Figure F.26: Multi-arch CPI plot for `lucas` (FP, F90, Number Theory)

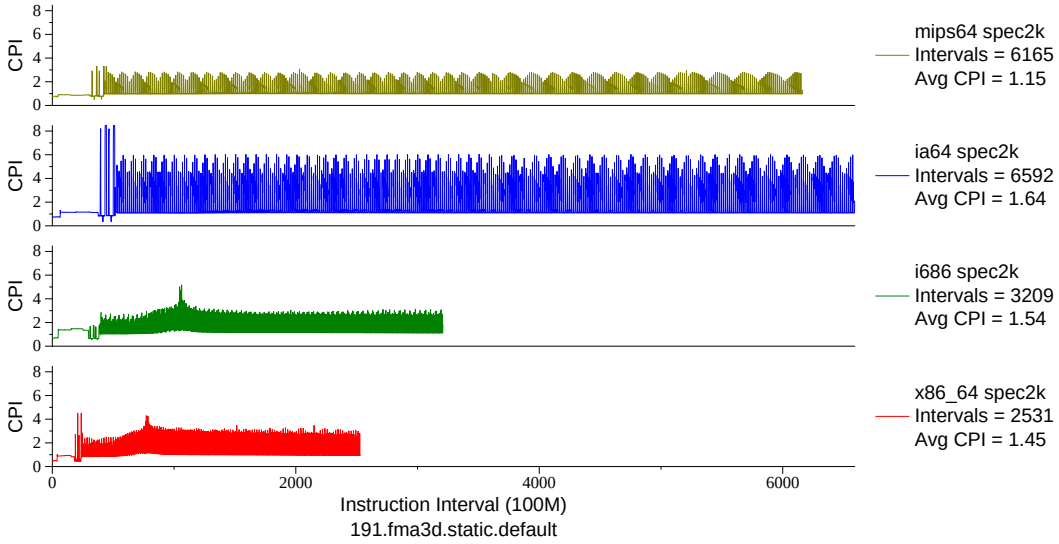

Figure F.27: Multi-arch CPI plot for `fma3d` (FP, F90, Crash Simulation)

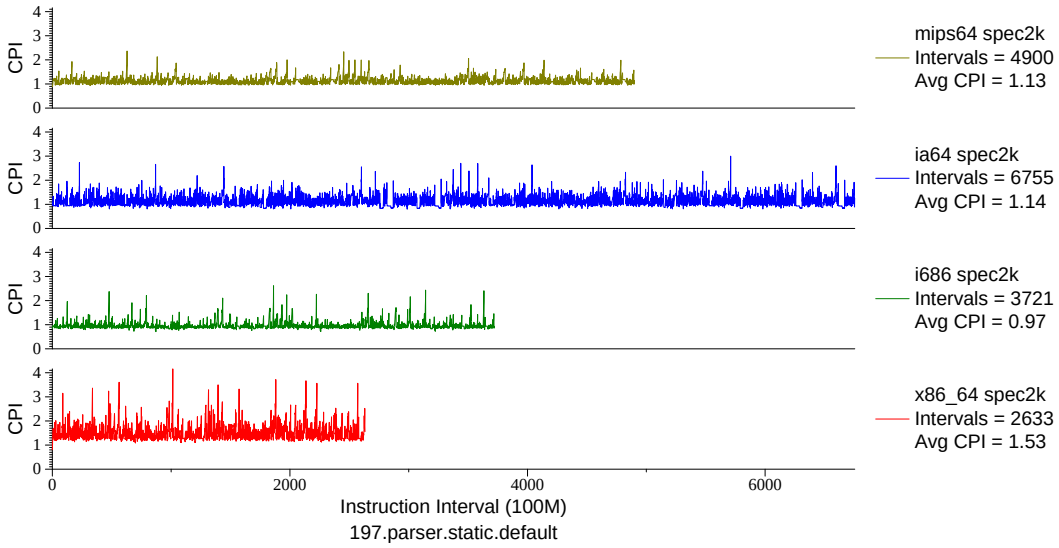

Figure F.28: Multi-arch CPI plot for `parser` (INT, C, Word Processing)

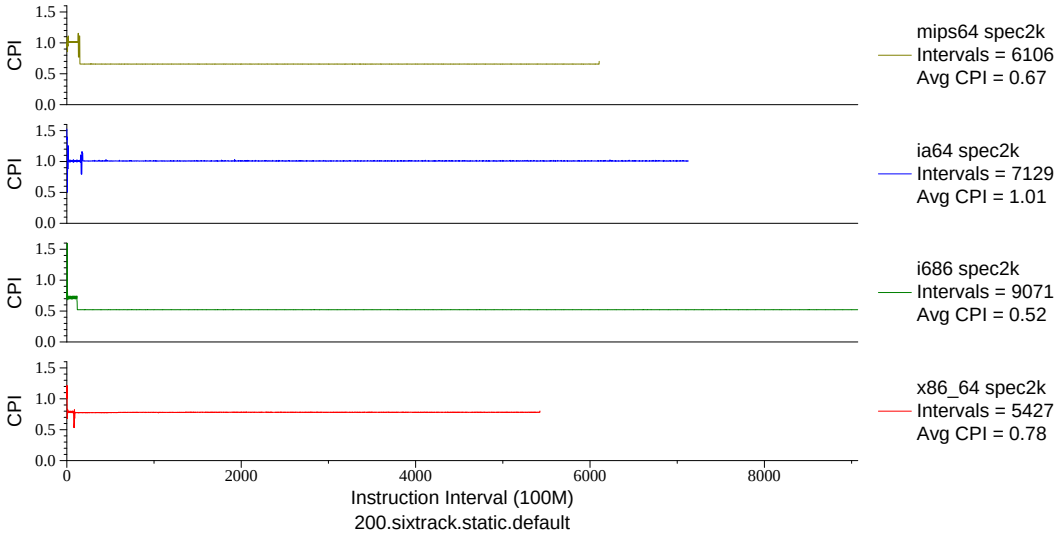

Figure F.29: Multi-arch CPI plot for `sixtrack` (FP, F77, Nuclear Physics)

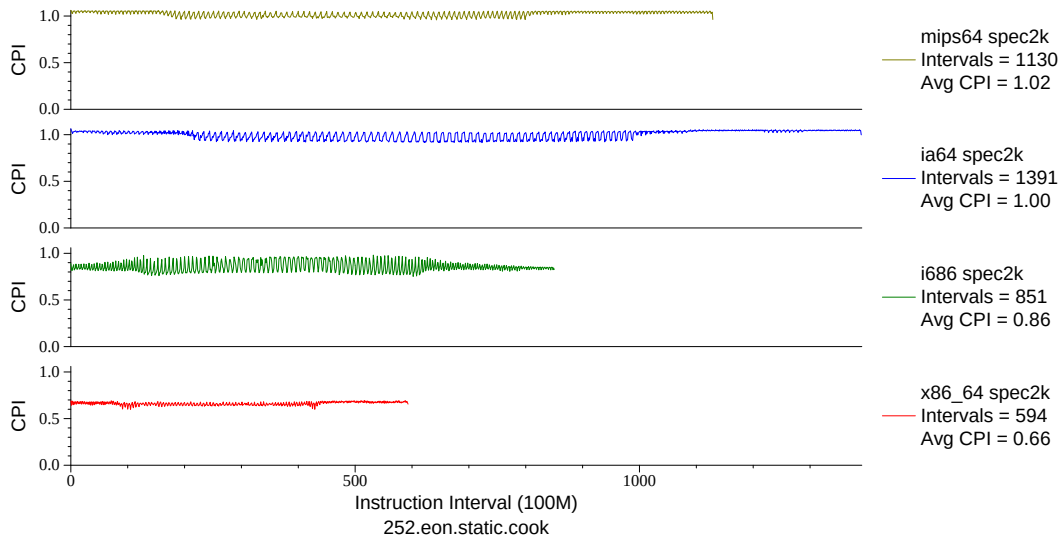

Figure F.30: Multi-arch CPI plot for `eon.cook` (INT, C++, Computer Graphics)

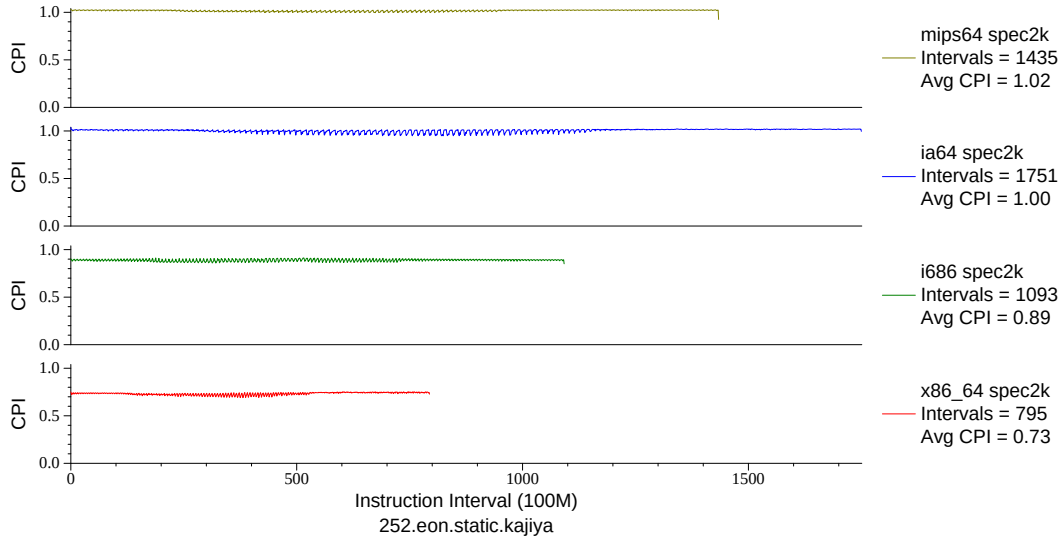

Figure F.31: Multi-arch CPI plot for `eon.kajiya` (INT, C++, Computer Graphics)

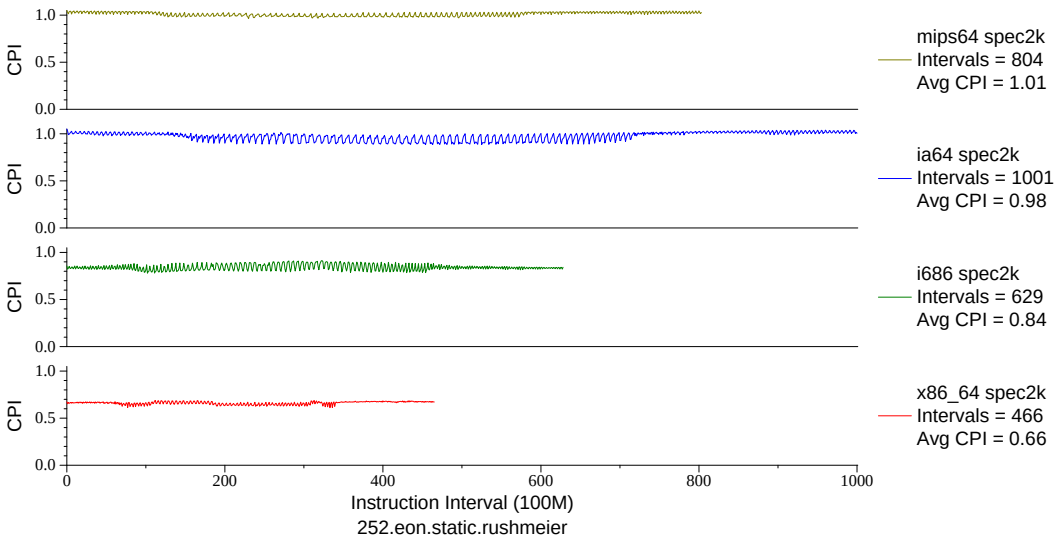

Figure F.32: Multi-arch CPI plot for `eon.rushmeier` (INT, C++, Computer Graphics)

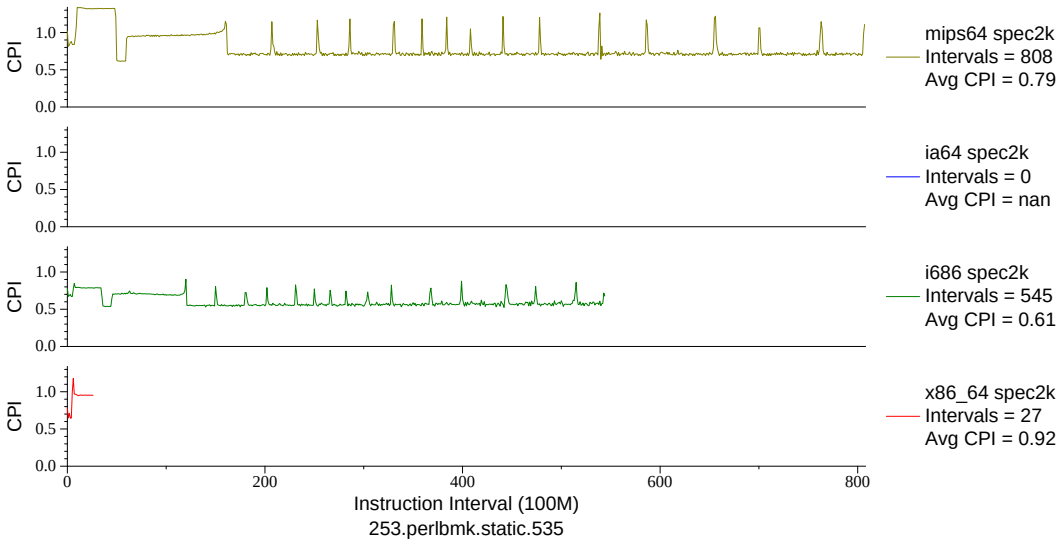

Figure F.33: Multi-arch CPI plot for `perlbmk.535` (INT, C, Scripting Language)

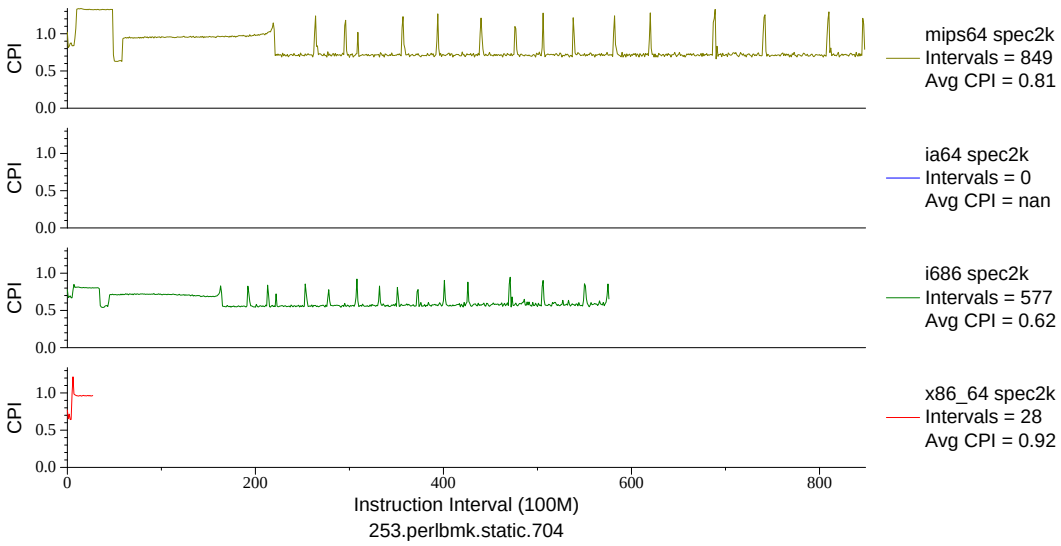

Figure F.34: Multi-arch CPI plot for `perlbmk.704` (INT, C, Scripting Language)

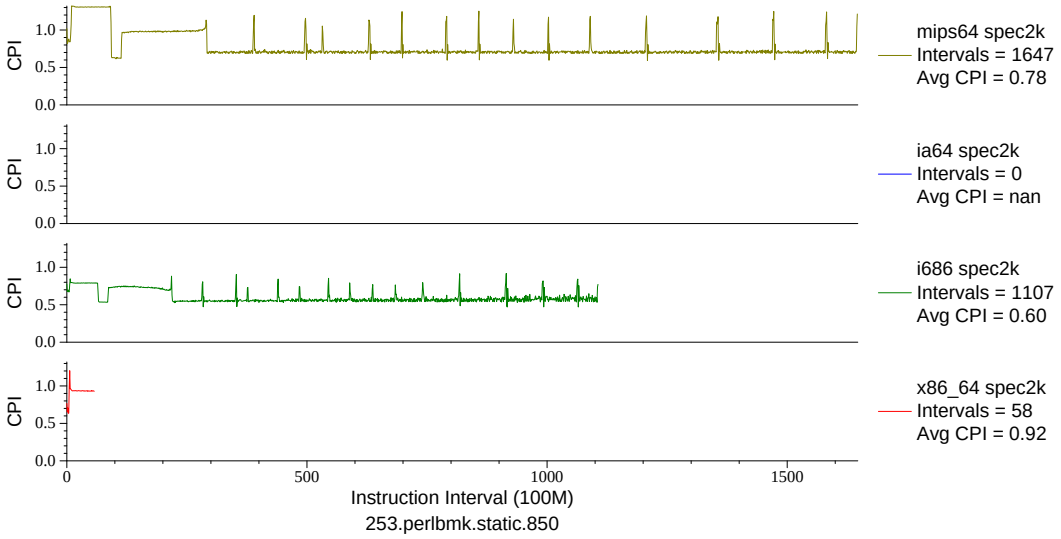

Figure F.35: Multi-arch CPI plot for `perlbmk.850` (INT, C, Scripting Language)

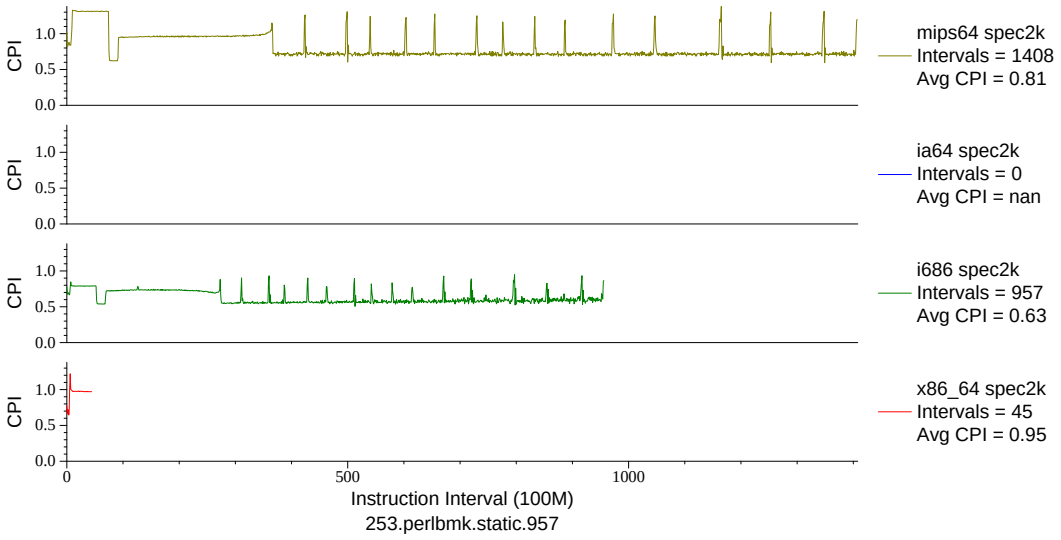

Figure F.36: Multi-arch CPI plot for `perlbmk.957` (INT, C, Scripting Language)

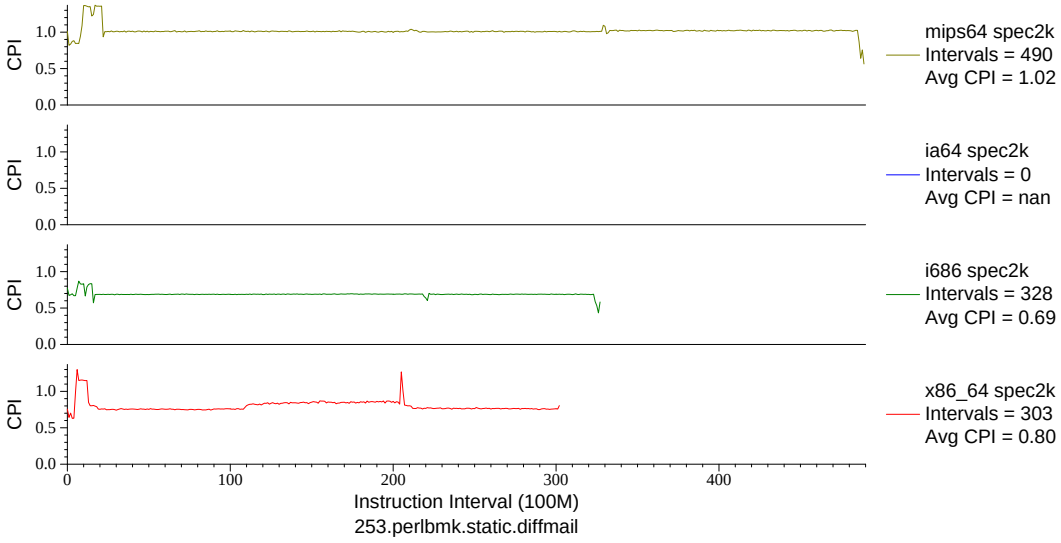

Figure F.37: Multi-arch CPI plot for `perlbmk.diff` (INT, C, Scripting Language)

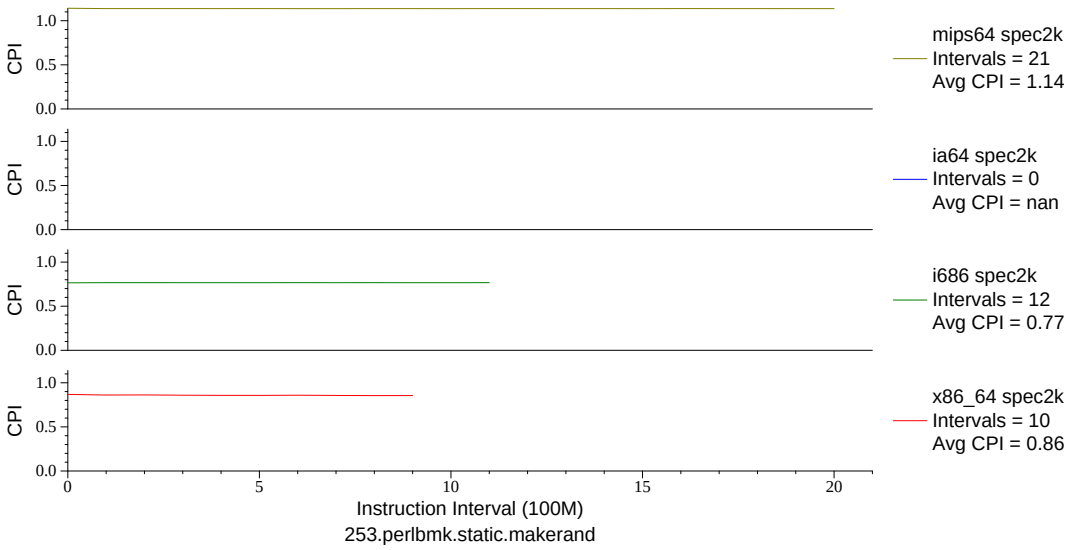

Figure F.38: Multi-arch CPI plot for `perlbmk.mkrnd` (INT, C, Scripting)

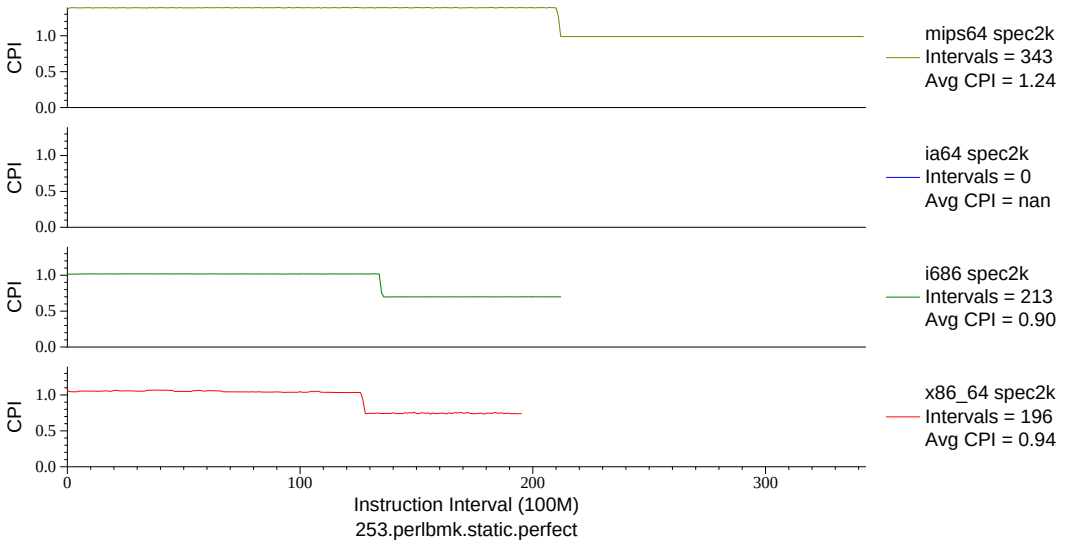

Figure F.39: Multi-arch CPI plot for `perlbmk.perf` (INT, C, Scripting)

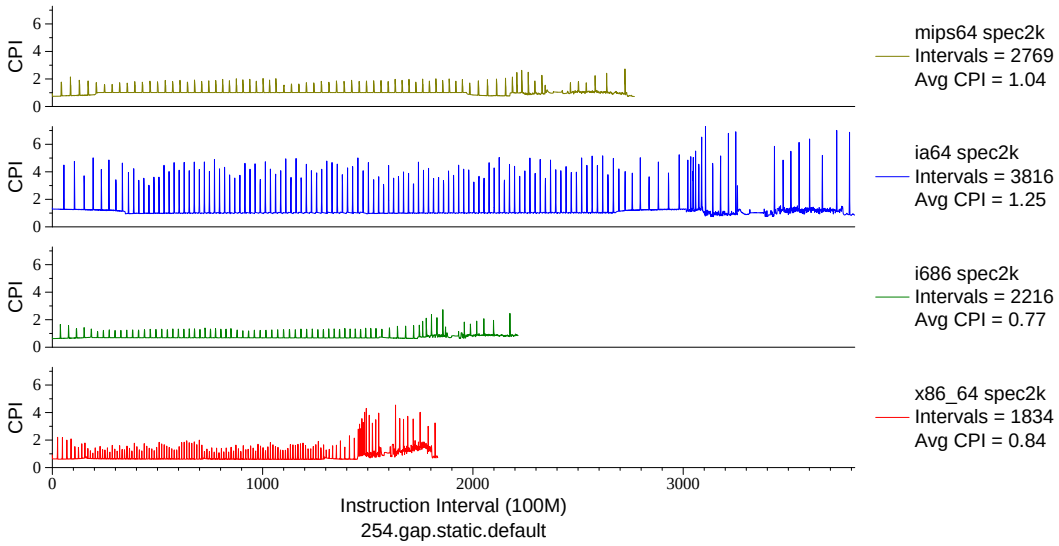

Figure F.40: Multi-arch CPI plot for `gap` (INT, C, Group Theory)

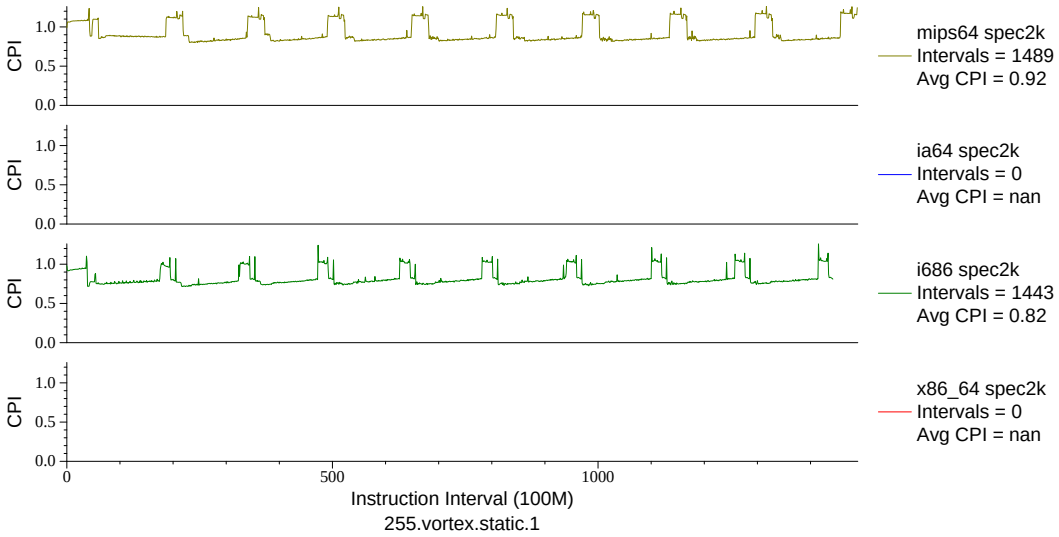

Figure F.41: Multi-arch CPI plot for `vortex.1` (INT, C, Database)

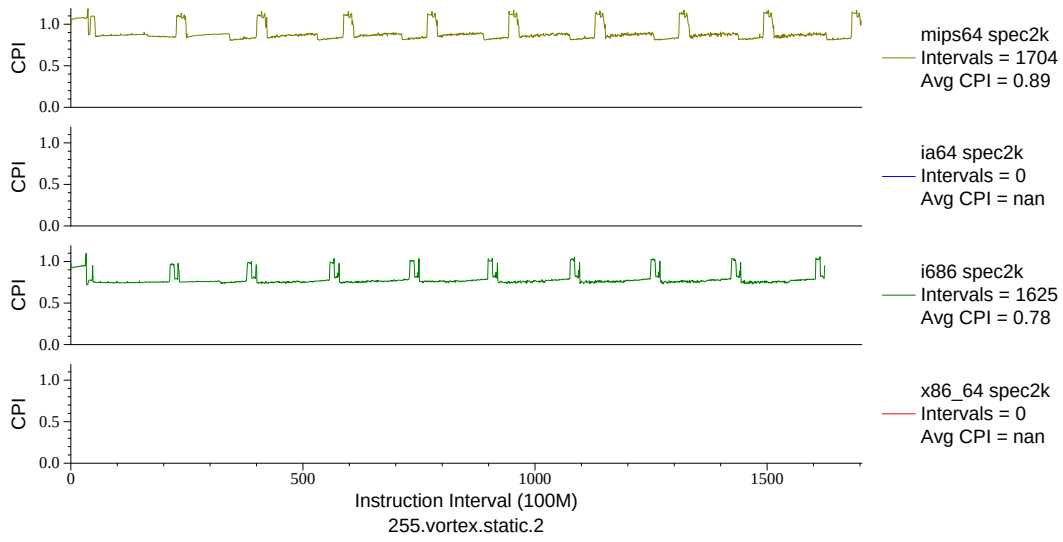

Figure F.42: Multi-arch CPI plot for `vortex.2` (INT, C, Database)

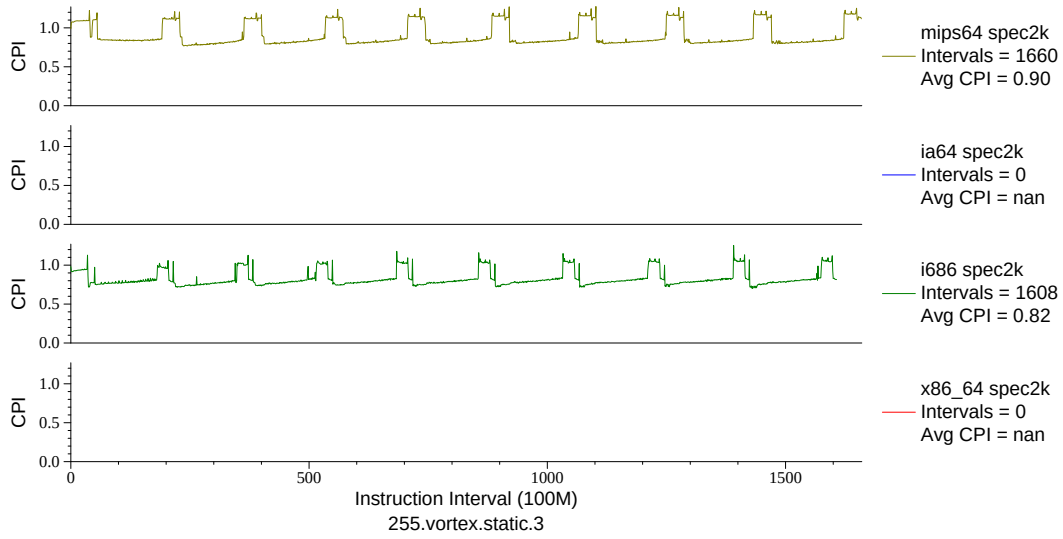

Figure F.43: Multi-arch CPI plot for `vortex.3` (INT, C, Database)

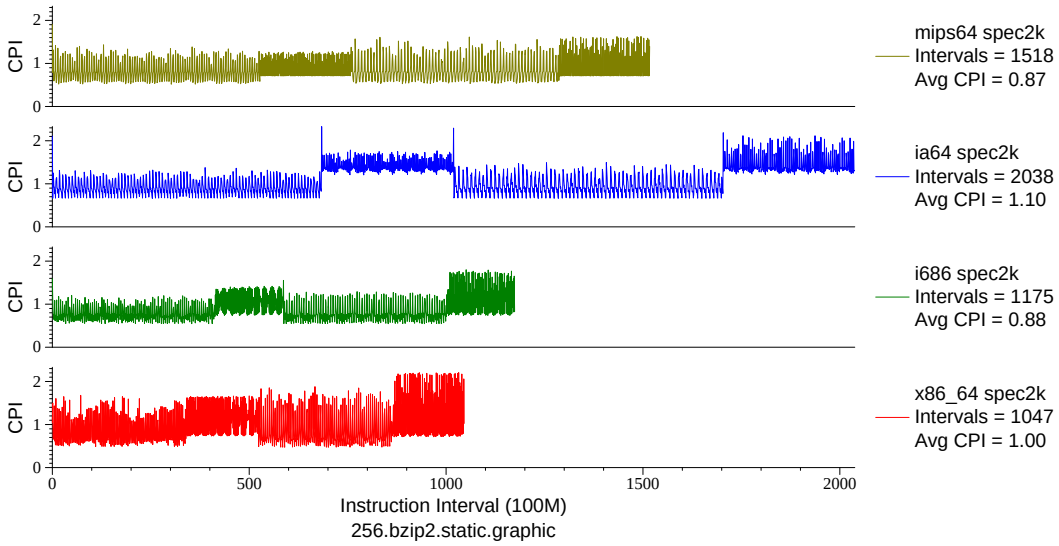

Figure F.44: Multi-arch CPI plot for `bzip2.graph` (INT, C, Compression)

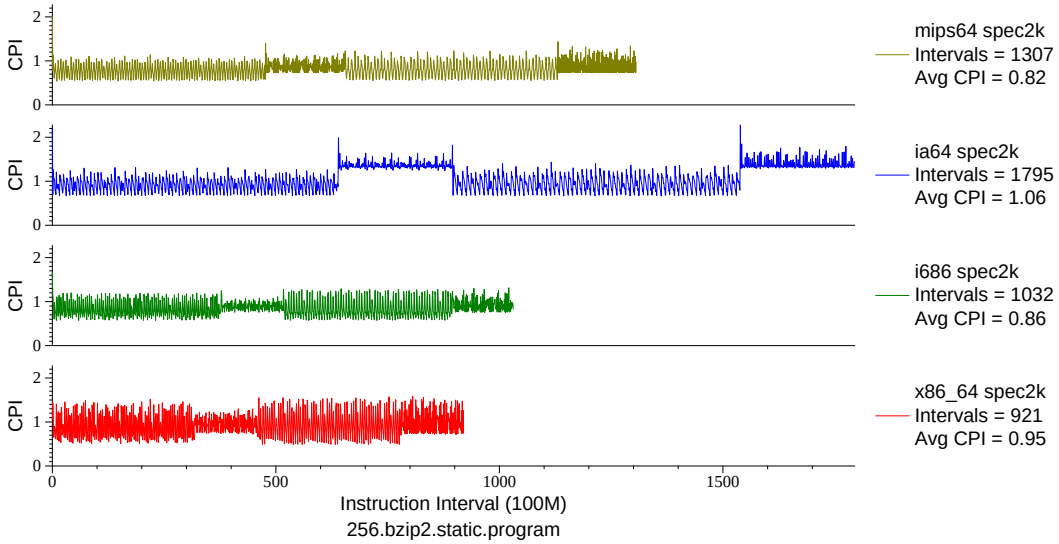

Figure F.45: Multi-arch CPI plot for `bzip2.prog` (INT, C, Compression)

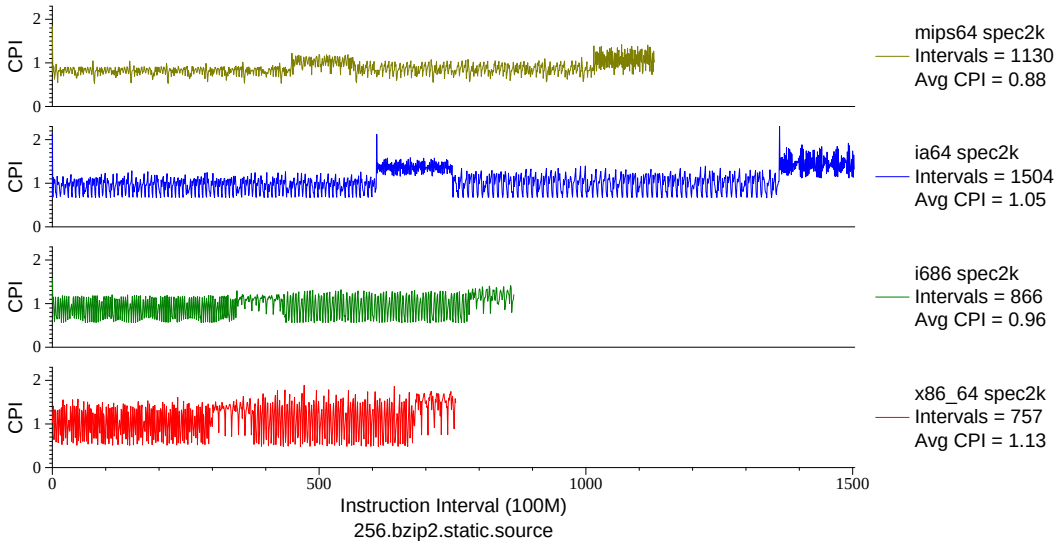

Figure F.46: Multi-arch CPI plot for `bzip2.src` (INT, C, Compression)

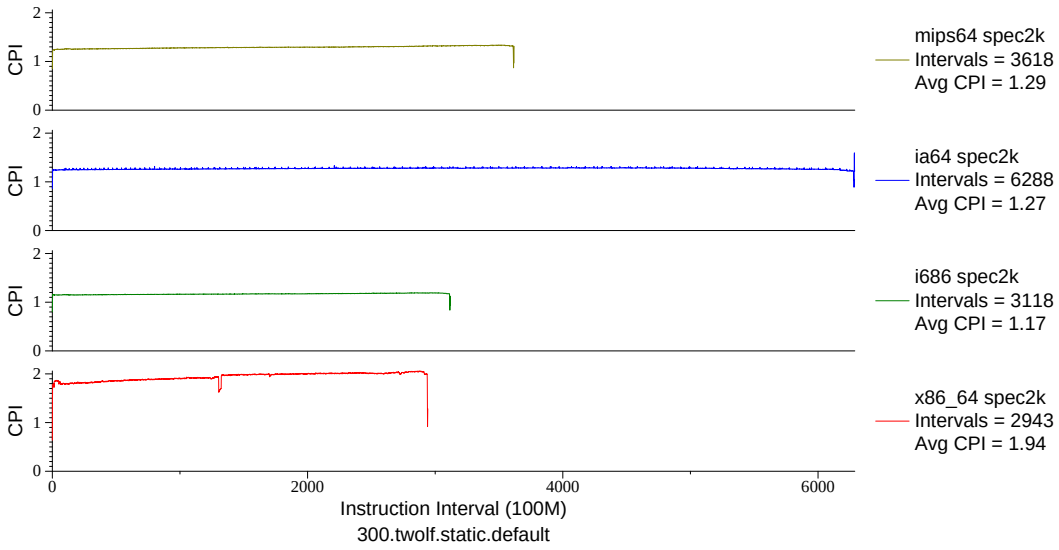

Figure F.47: Multi-arch CPI plot for `twolf` (INT, C, Place/Route)

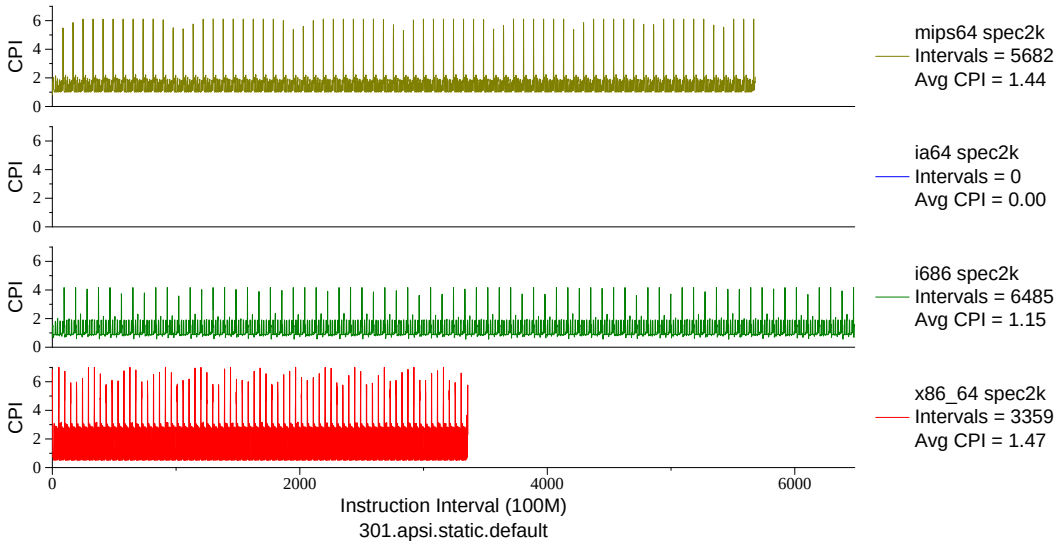

Figure F.48: Multi-arch CPI plot for `apsi` (FP, F77, Meteorology/Pollution)

APPENDIX G

L1 DATA CACHE ACCESSES PER INSTRUCTION PHASE PLOTS

When investigating memory subsystems it is important that your simulation method creates a faithful representation of the processor's memory access patterns. One way to measure this is L1 data cache accesses per retired instruction (DPI). The following figures contain phase plots showing data cache accesses per instruction for SPEC CPU2000 on three actual x86_64 machines. Results from the SimPoint, unguided fast-forwarding, and start from the beginning reduced execution methods are also shown. Simulator results for m5 and Valgrind are included for comparison, as are results from the MIPS architecture.

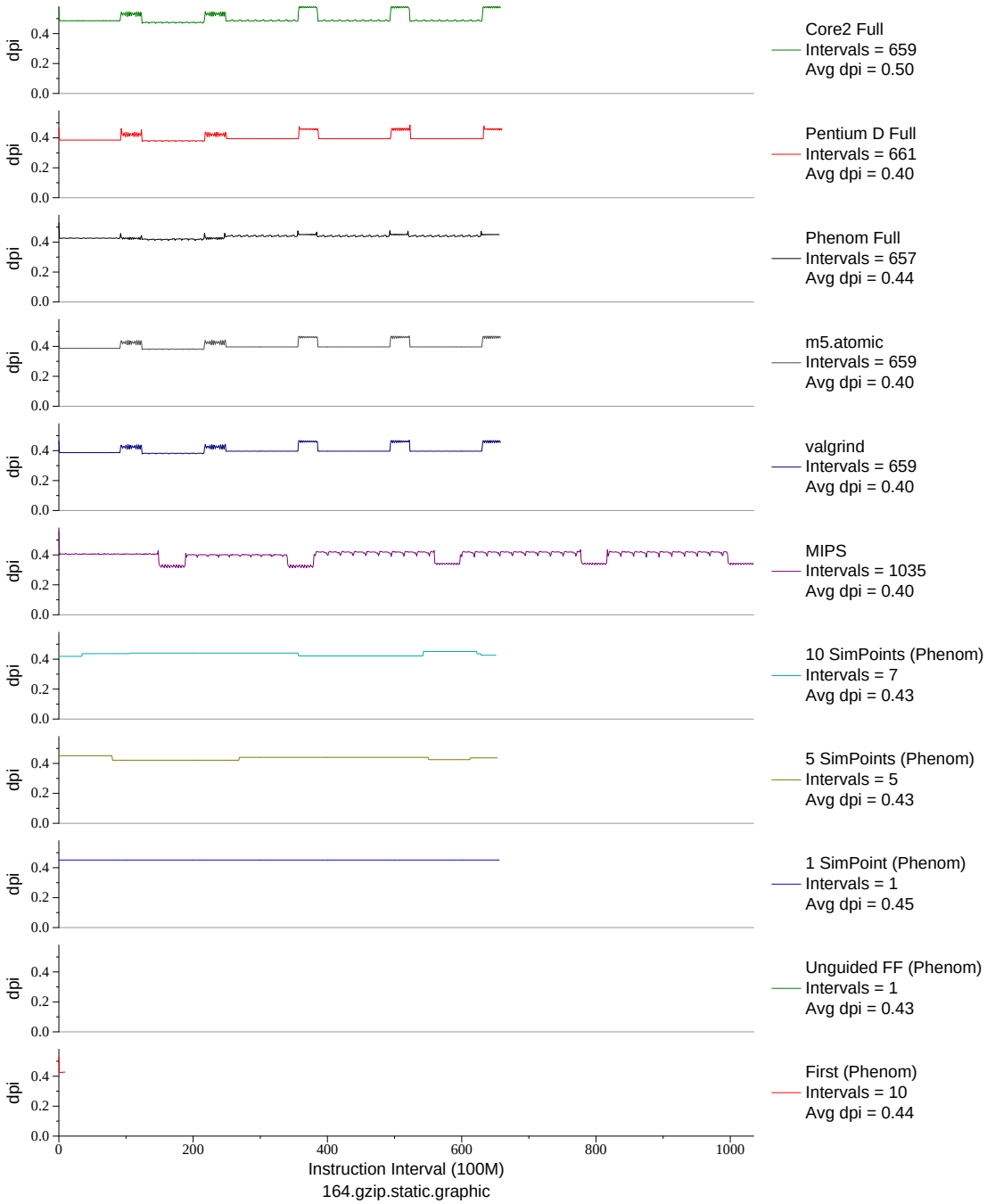

Figure G.1: L1 dcache accesses per instruction plot for `gzip.graph` (INT, C, Compression)

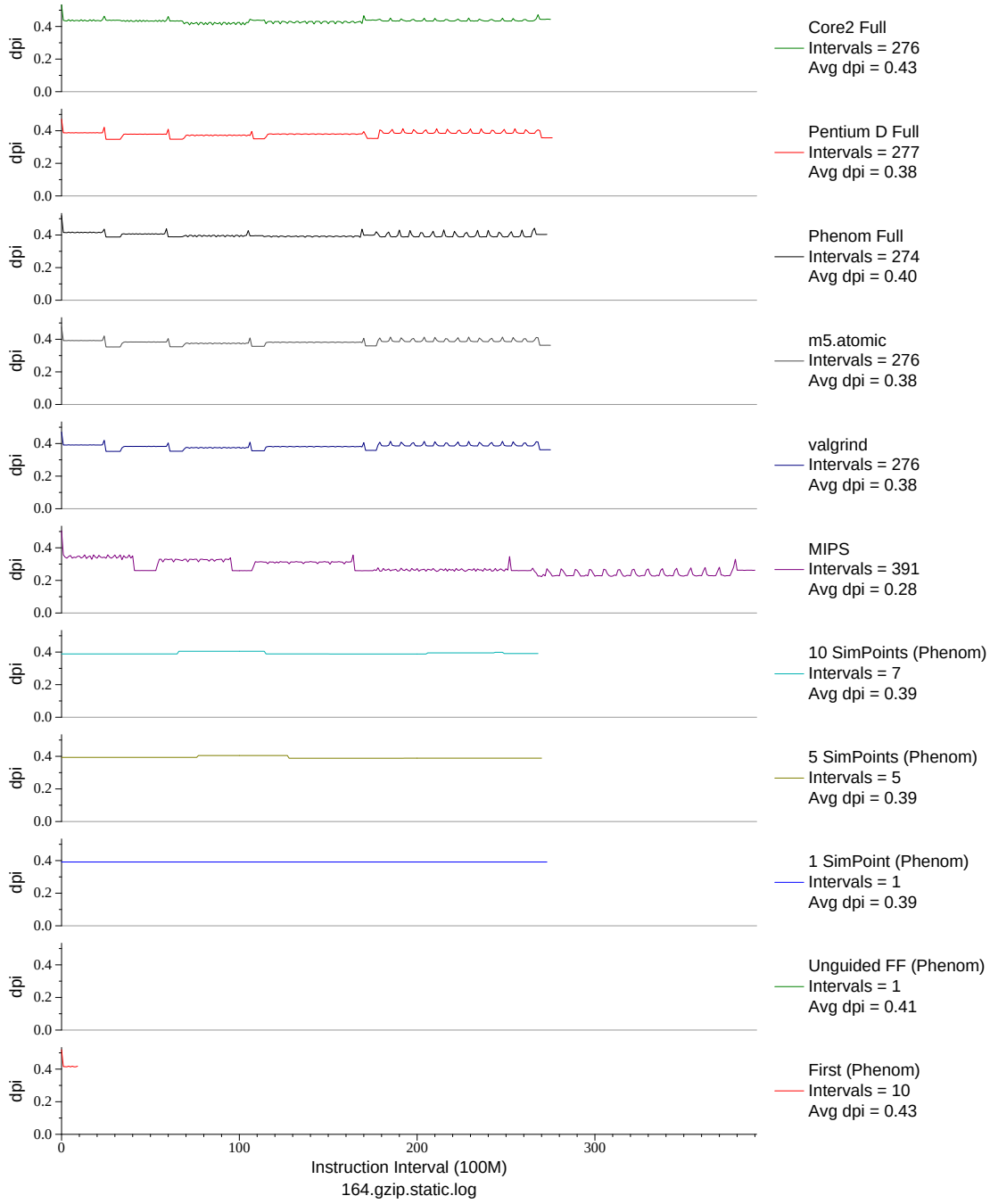

Figure G.2: L1 dcache accesses per instruction plot for `gzip.log` (INT, C, Compression)

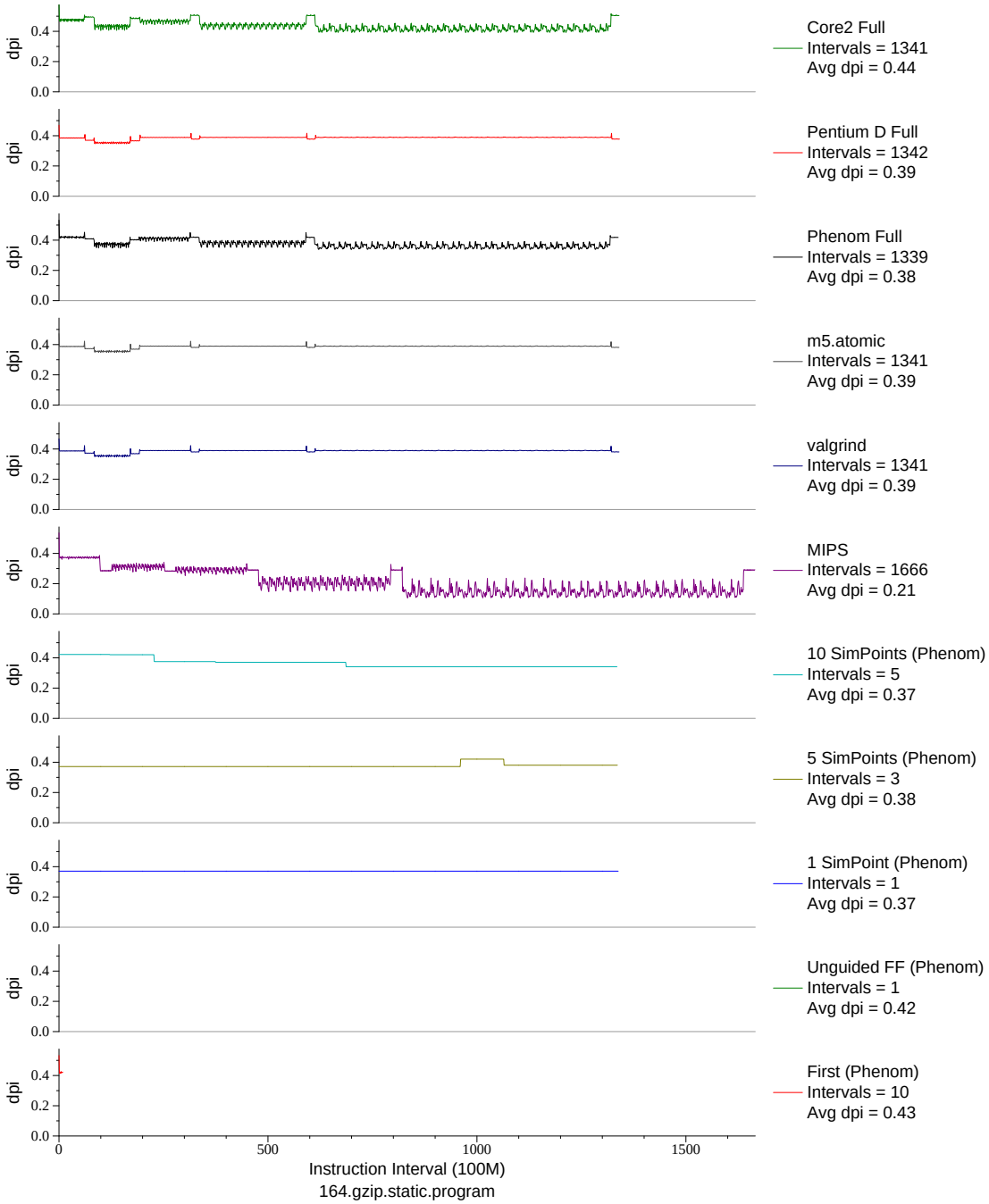

Figure G.3: L1 dcache accesses per instruction plot for `gzip.prog` (INT, C, Compression)

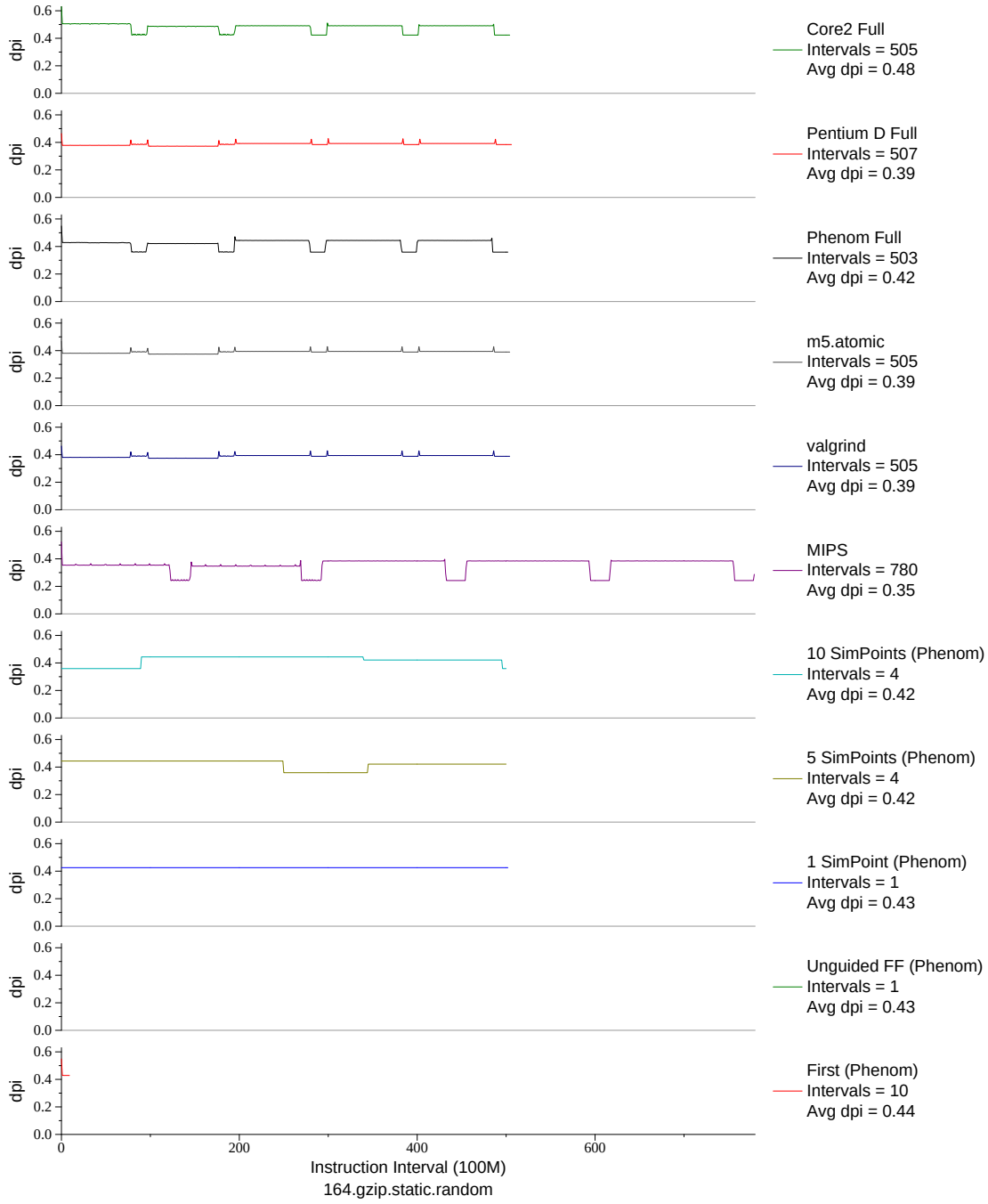

Figure G.4: L1 dcache accesses per instruction plot for `gzip.rand` (INT, C, Compression)

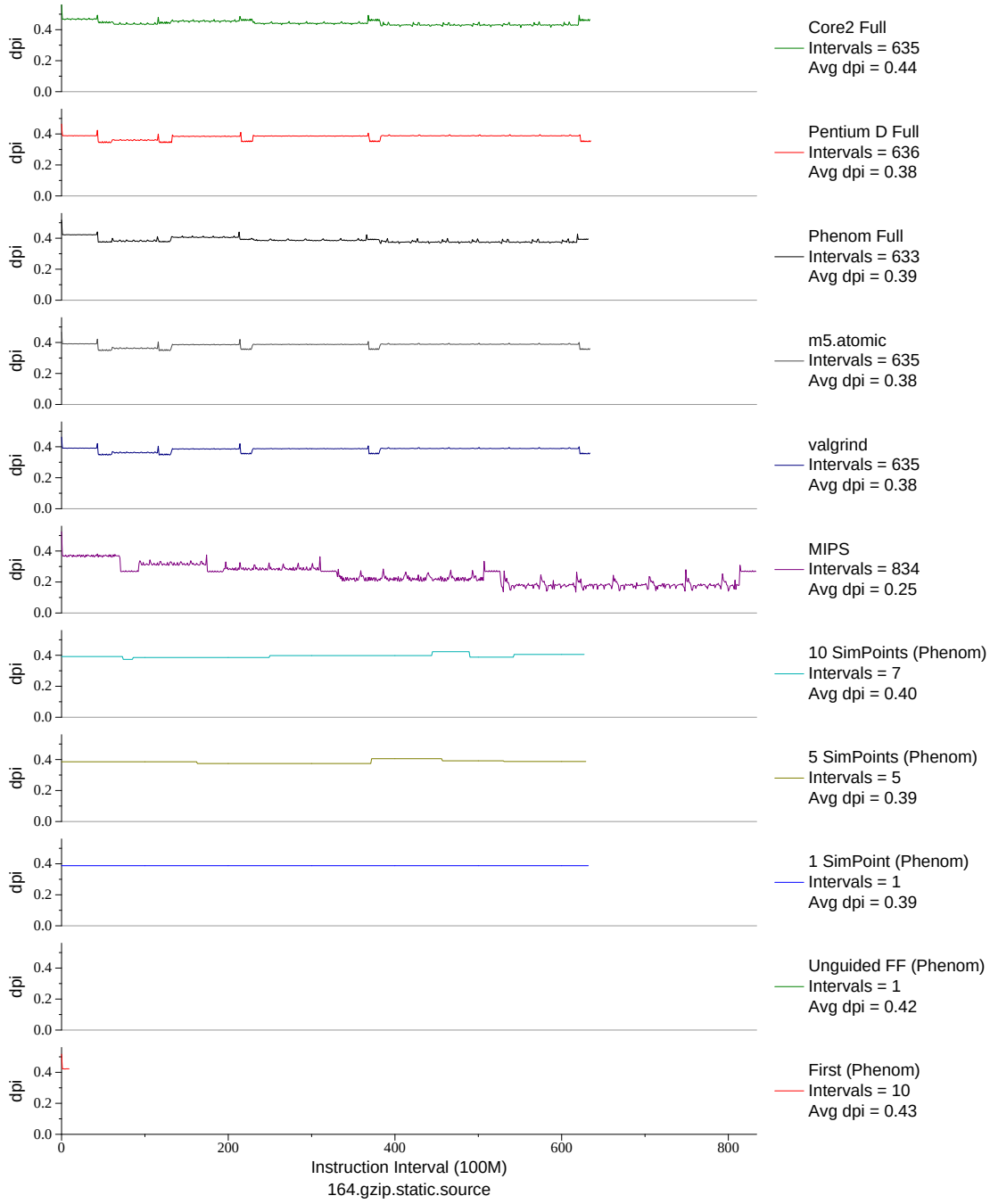

Figure G.5: L1 dcache accesses per instruction plot for `gzip.src` (INT, C, Compression)

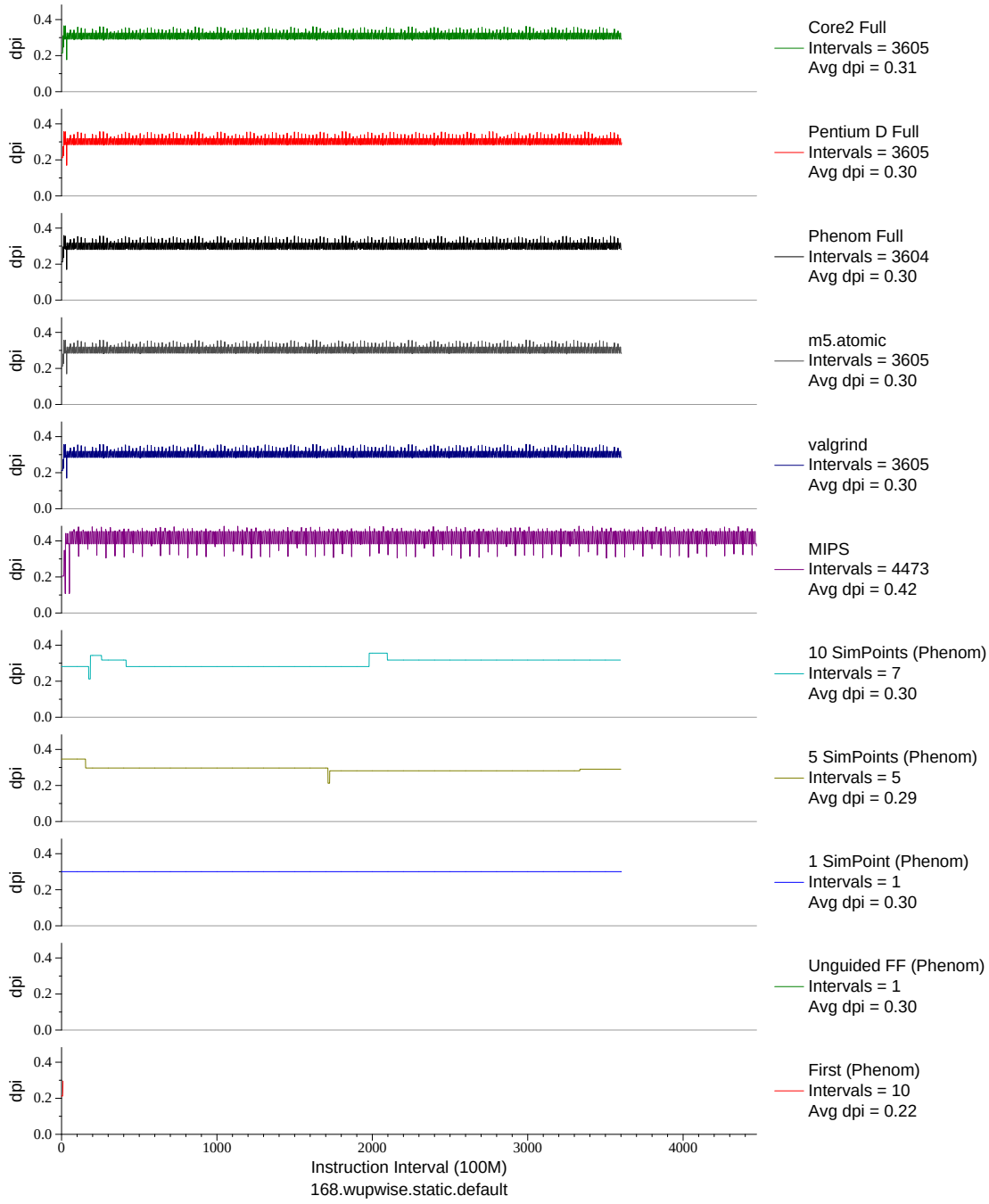

Figure G.6: L1 dcache accesses per instruction plot for `wupwise` (FP, F77, Quantum Chromodynamics)

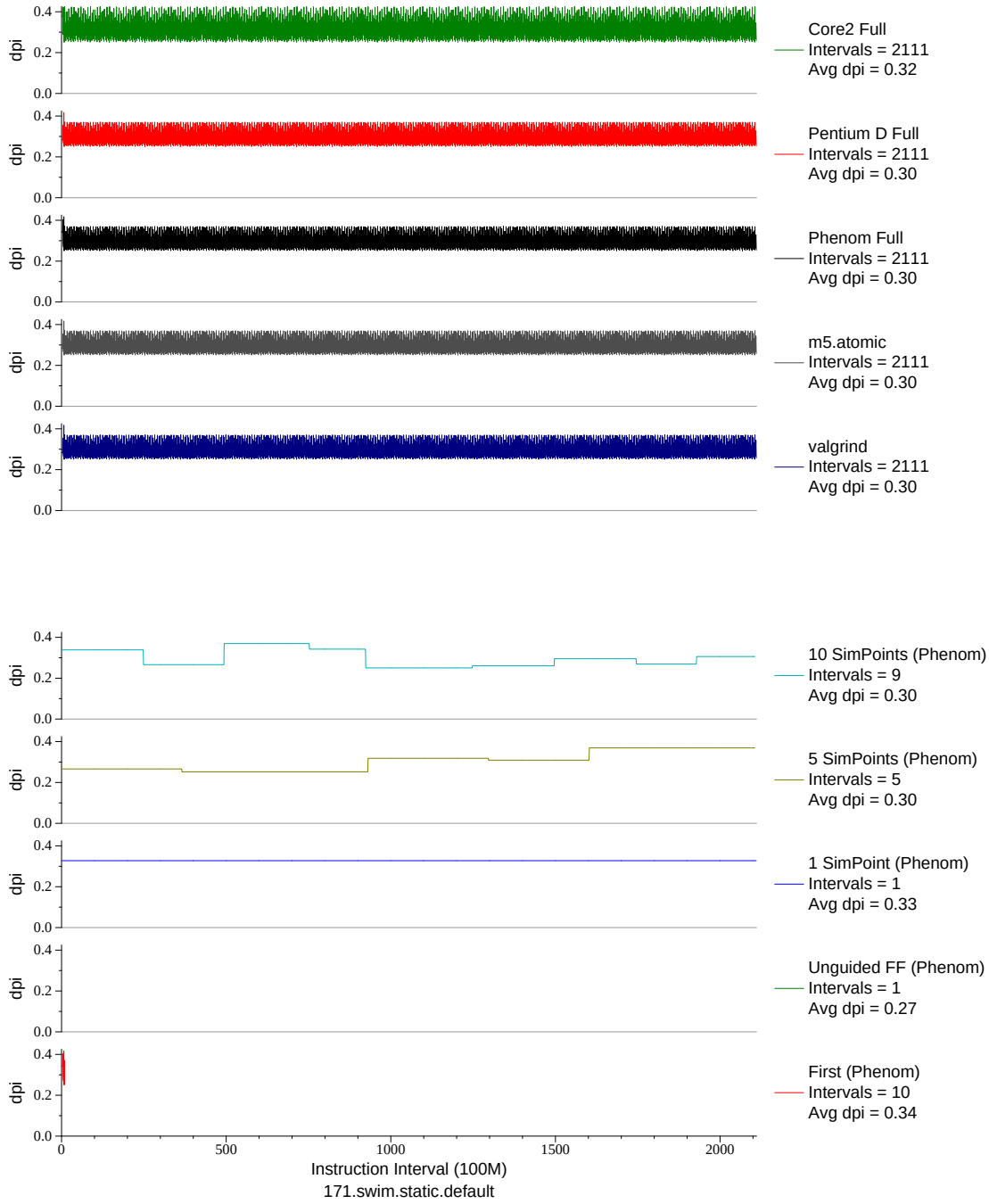

Figure G.7: L1 dcache accesses per instruction plot for `swim` (FP, F77, Meteorology/Water)

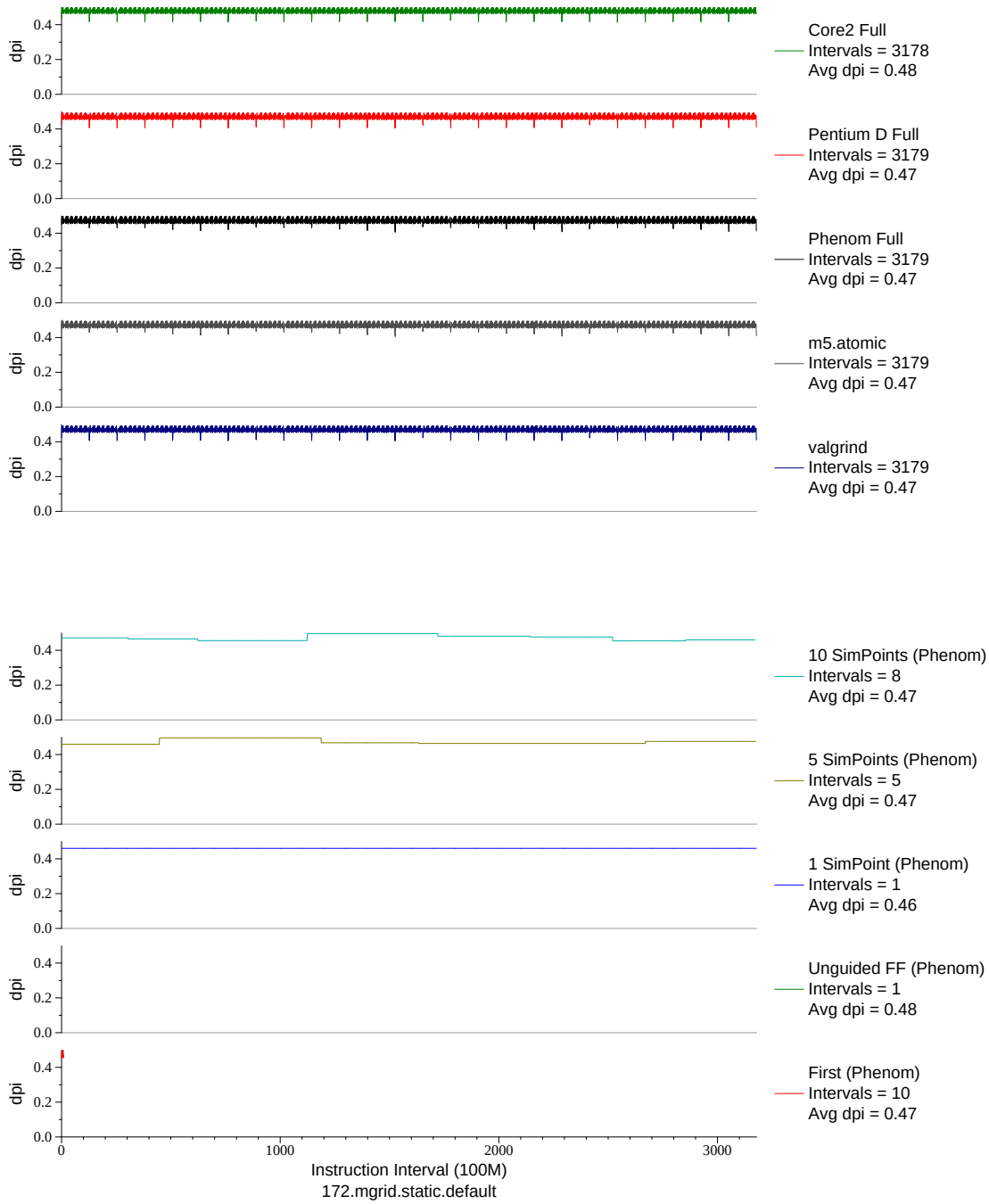

Figure G.8: L1 dcache accesses per instruction plot for `mgrid` (FP, F77, Multi-Grid Solver)

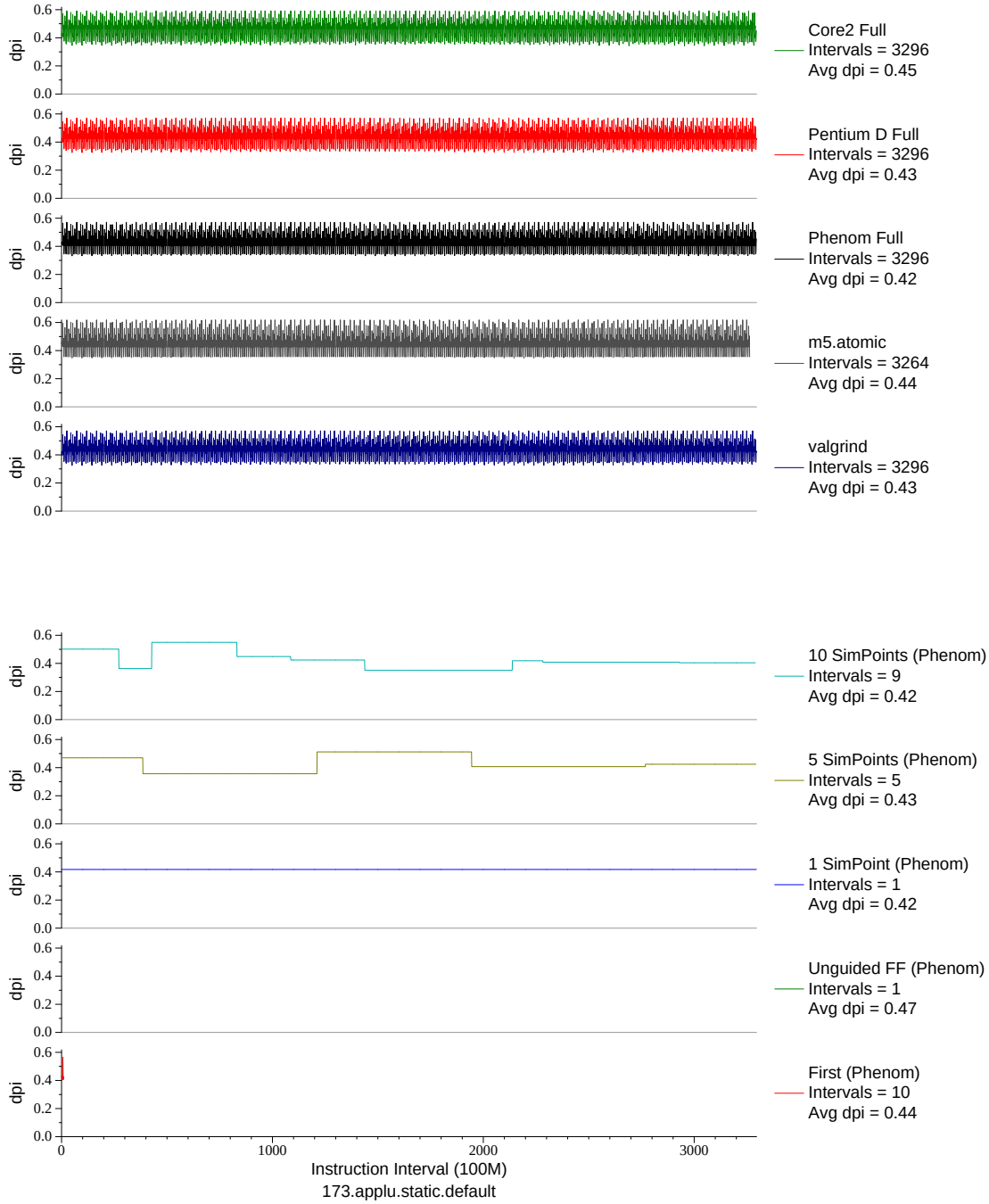

Figure G.9: L1 dcache accesses per instruction plot for `applu` (FP, F77, Fluid Dynamics)

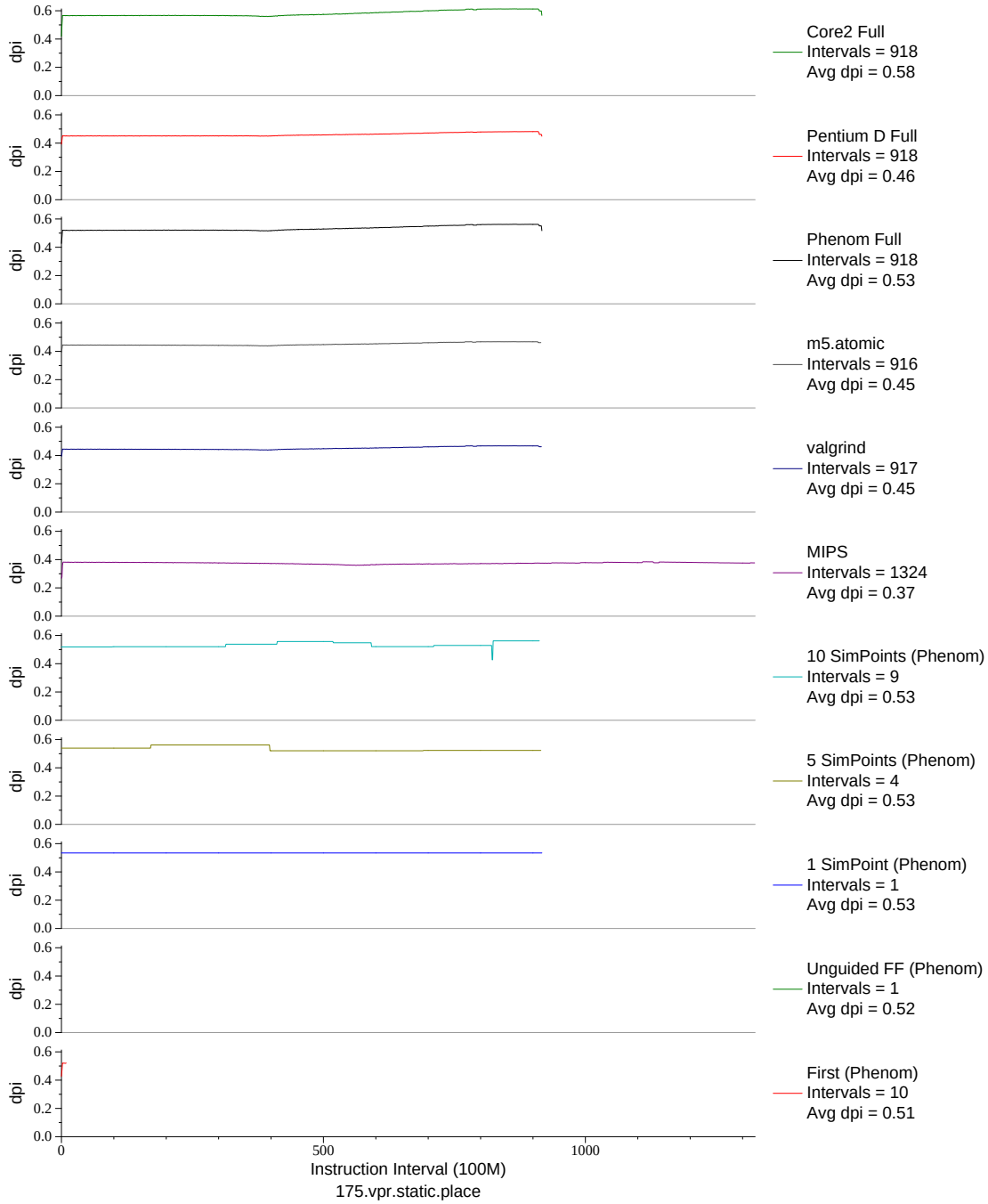

Figure G.10: L1 dcache accesses per instruction plot for `vpr.place` (INT, C, FPGA Place/Route)

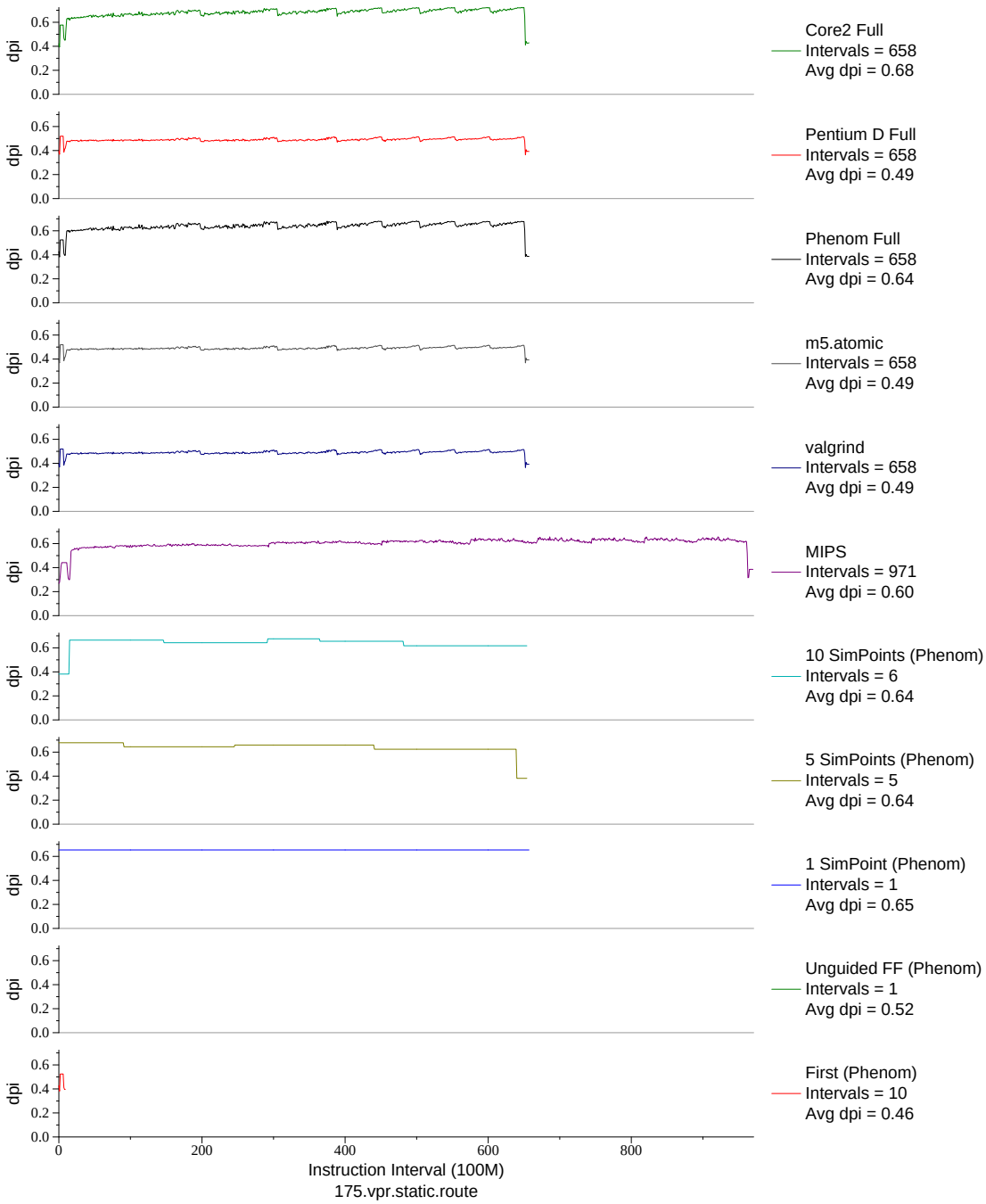

Figure G.11: L1 dcache accesses per instruction plot for `vpr.route` (INT, C, FPGA Place/Route)

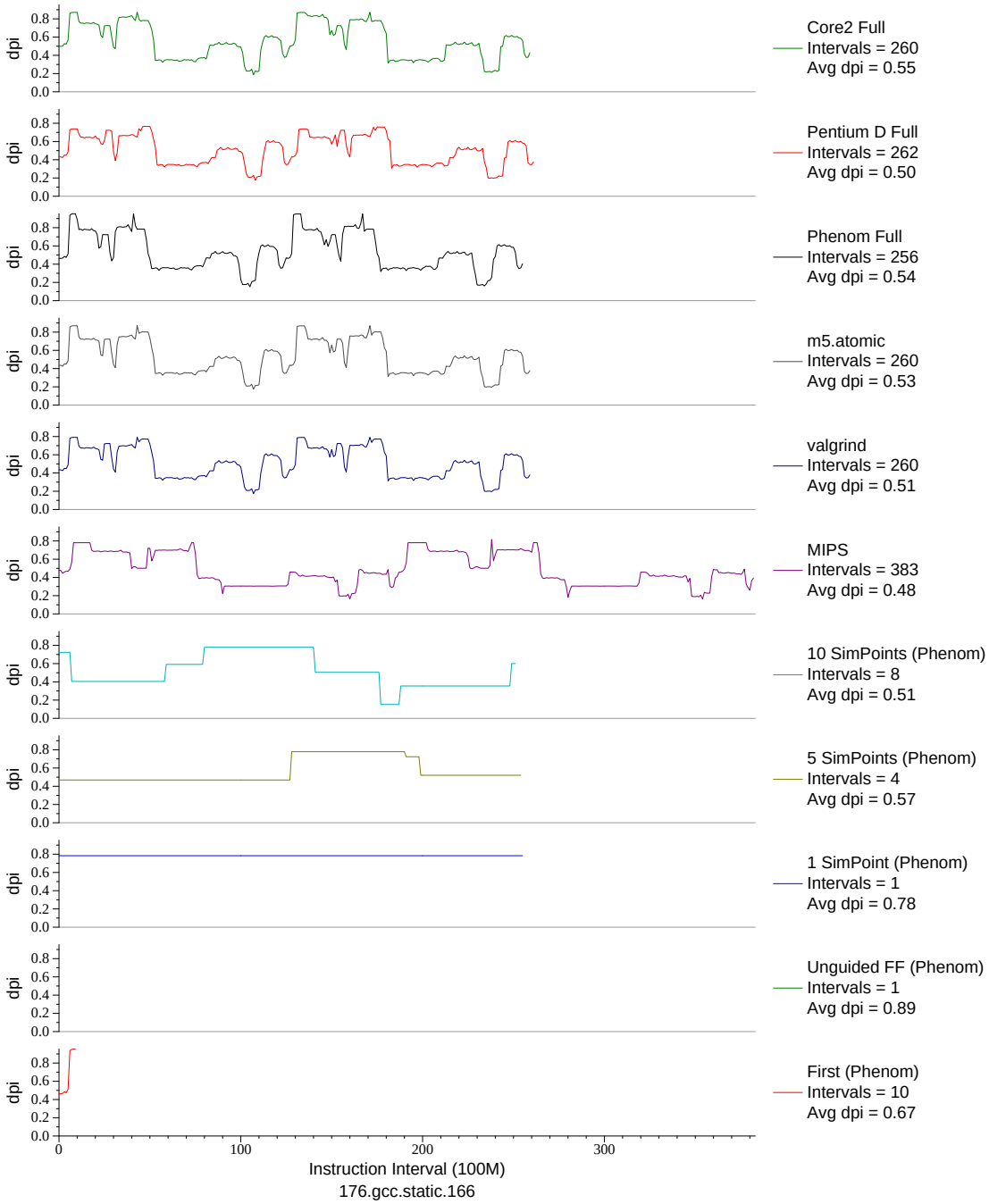

Figure G.12: L1 dcache accesses per instruction plot for `gcc.166` (INT, C, C Compiler)

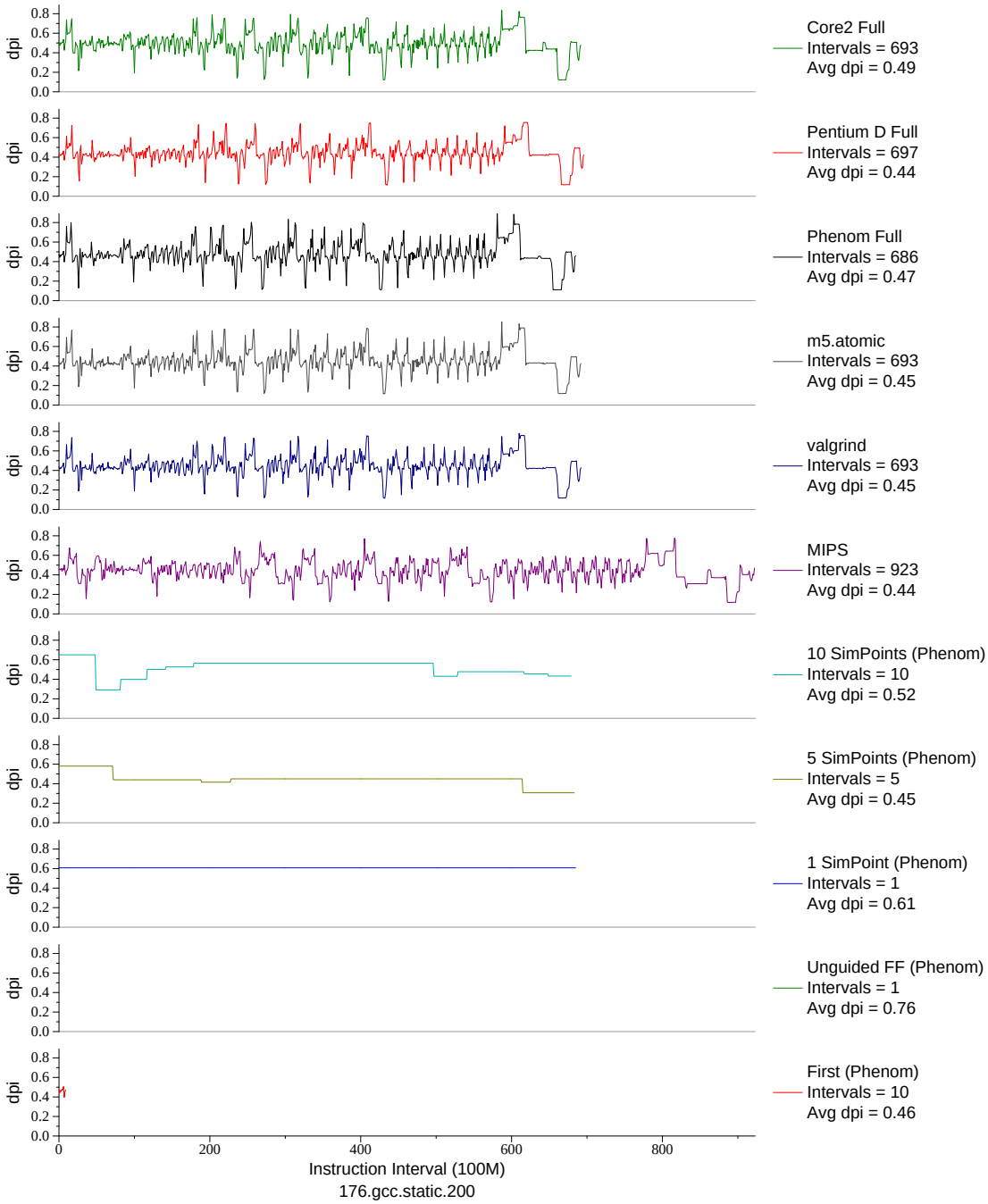

Figure G.13: L1 dcache accesses per instruction plot for `gcc.200` (INT, C, C Compiler)

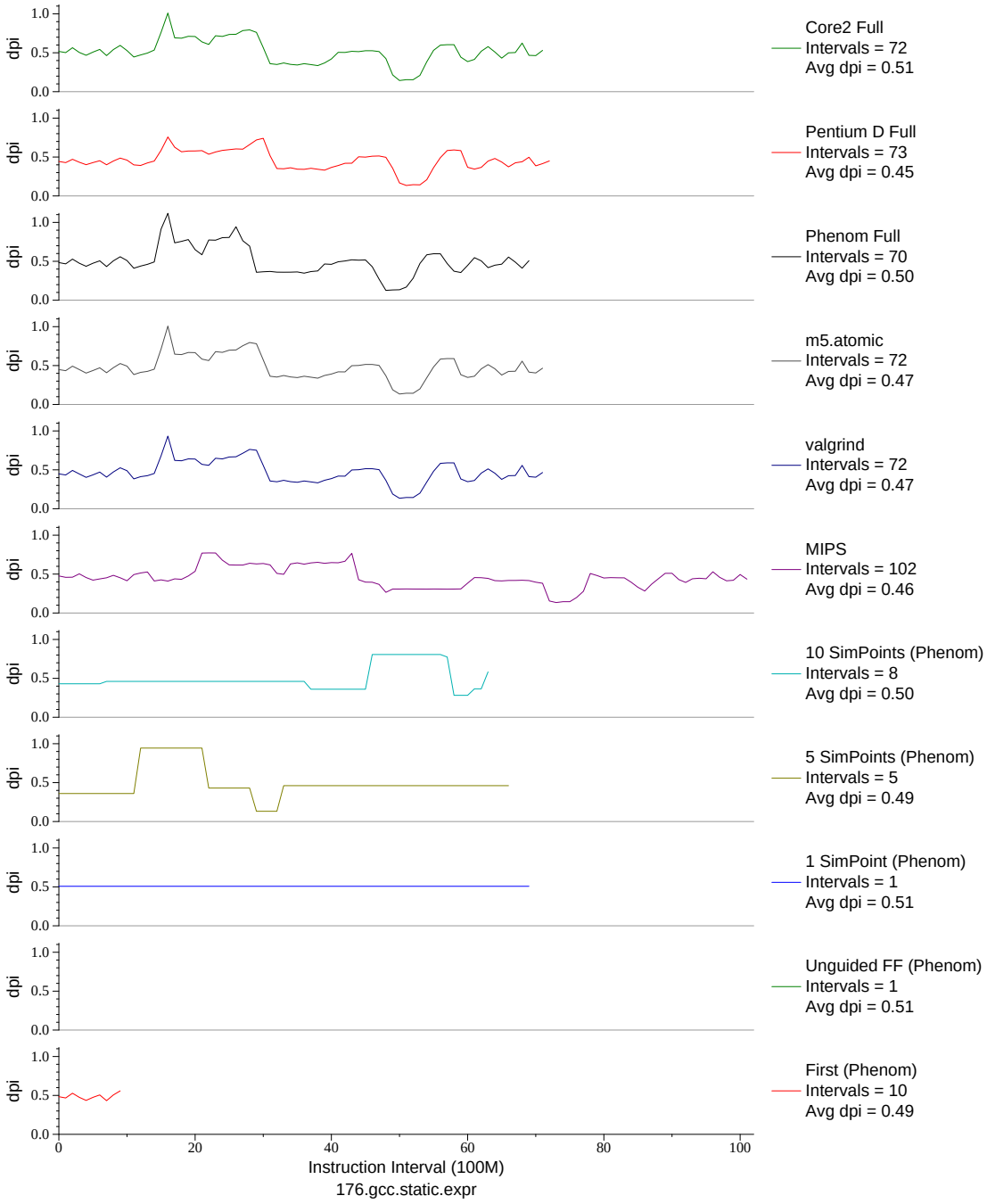

Figure G.14: L1 dcache accesses per instruction plot for `gcc.expr` (INT, C, C Compiler)

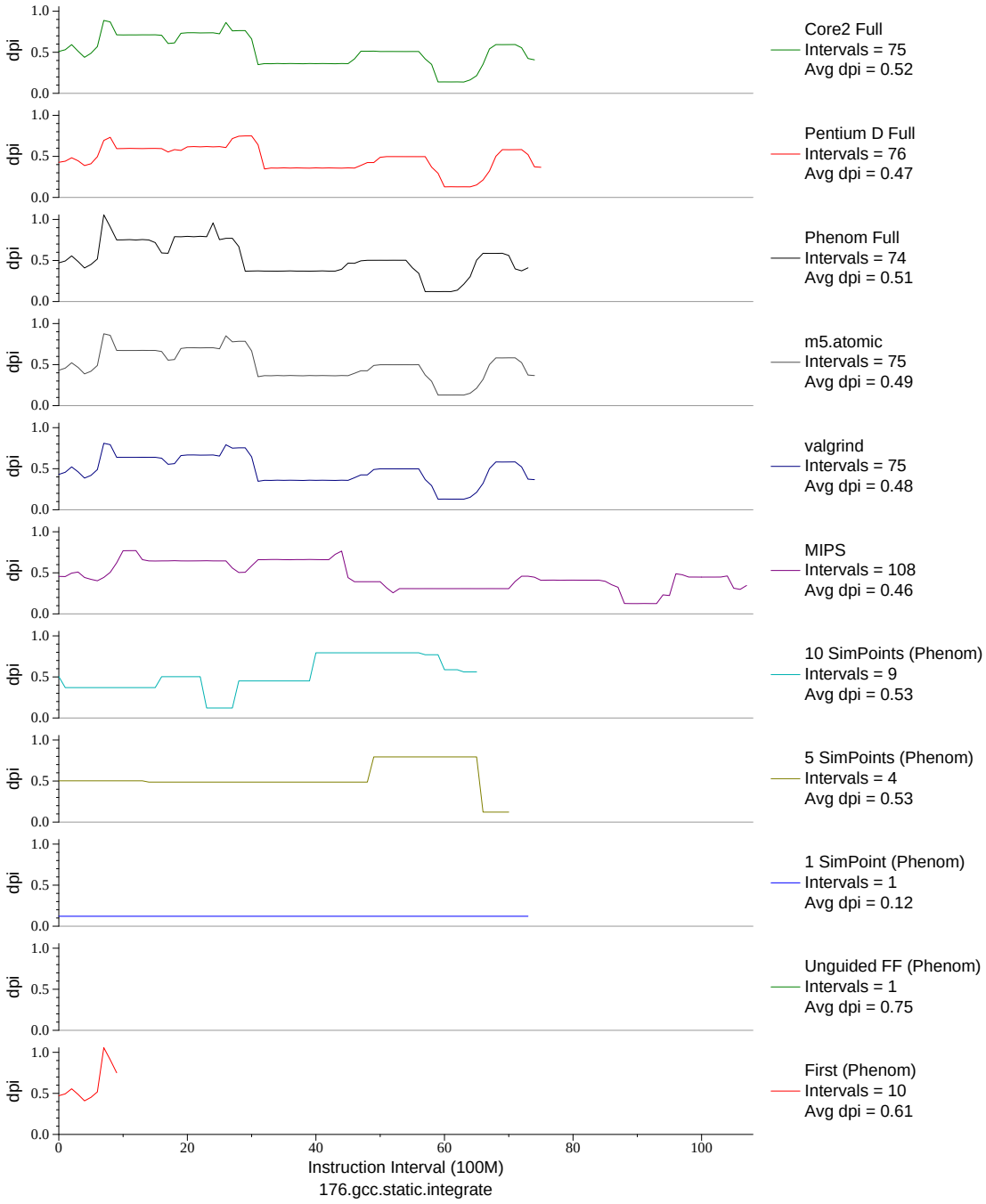

Figure G.15: L1 dcache accesses per instruction plot for `gcc.int` (INT, C, C Compiler)

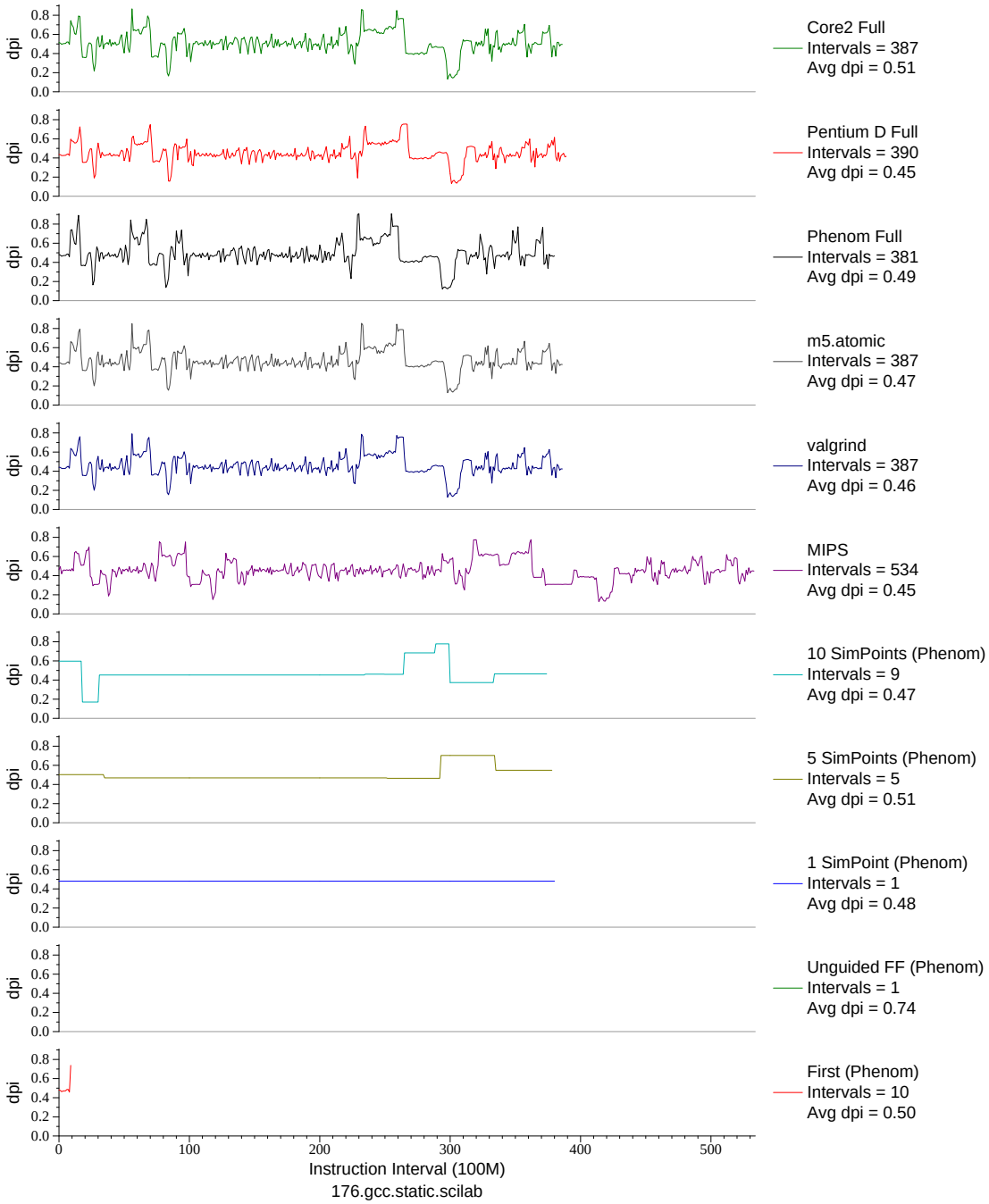

Figure G.16: L1 dcache accesses per instruction plot for `gcc.sci` (INT, C, C Compiler)

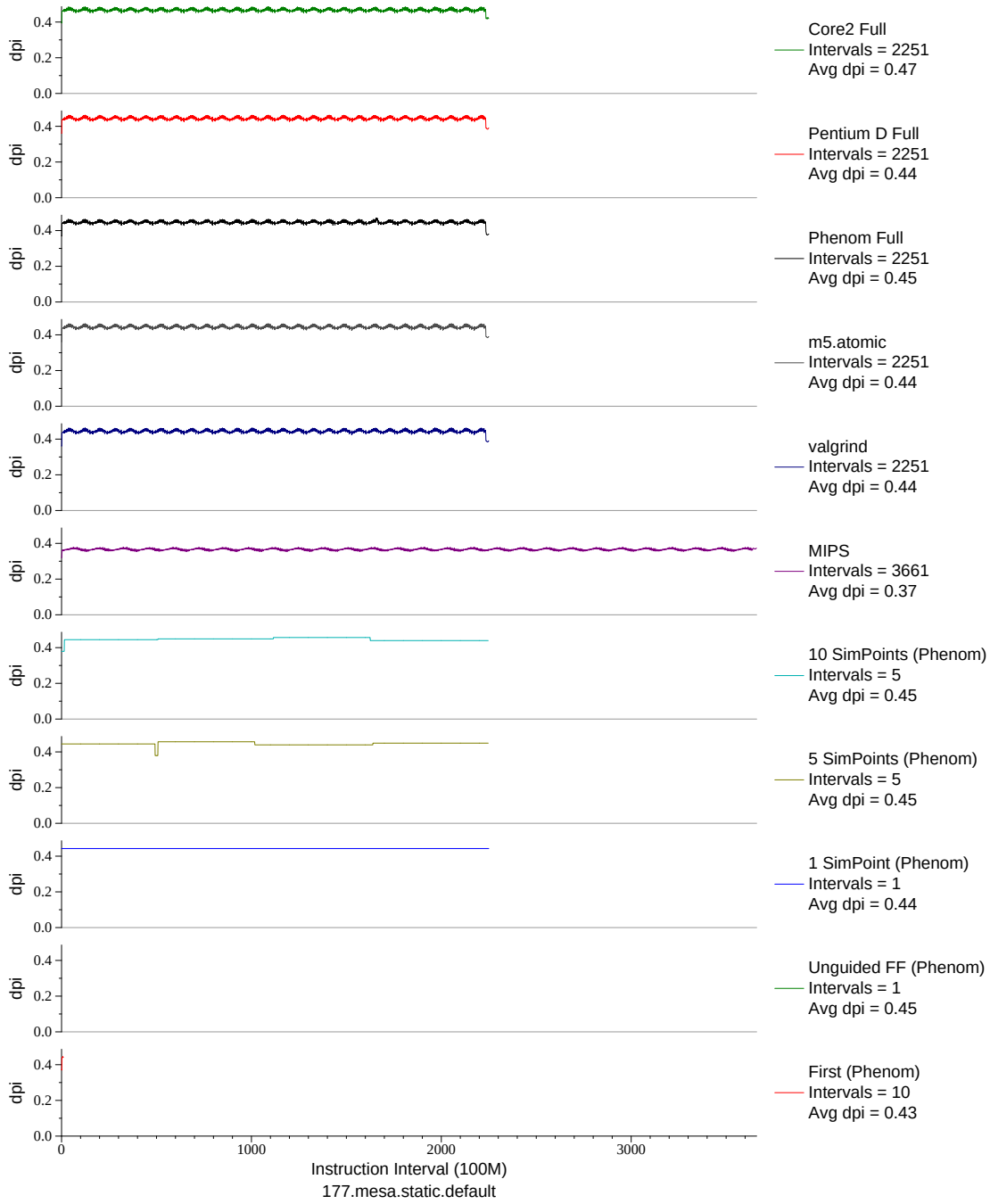

Figure G.17: L1 dcache accesses per instruction plot for `mesa` (FP, C, 3D-graphics)

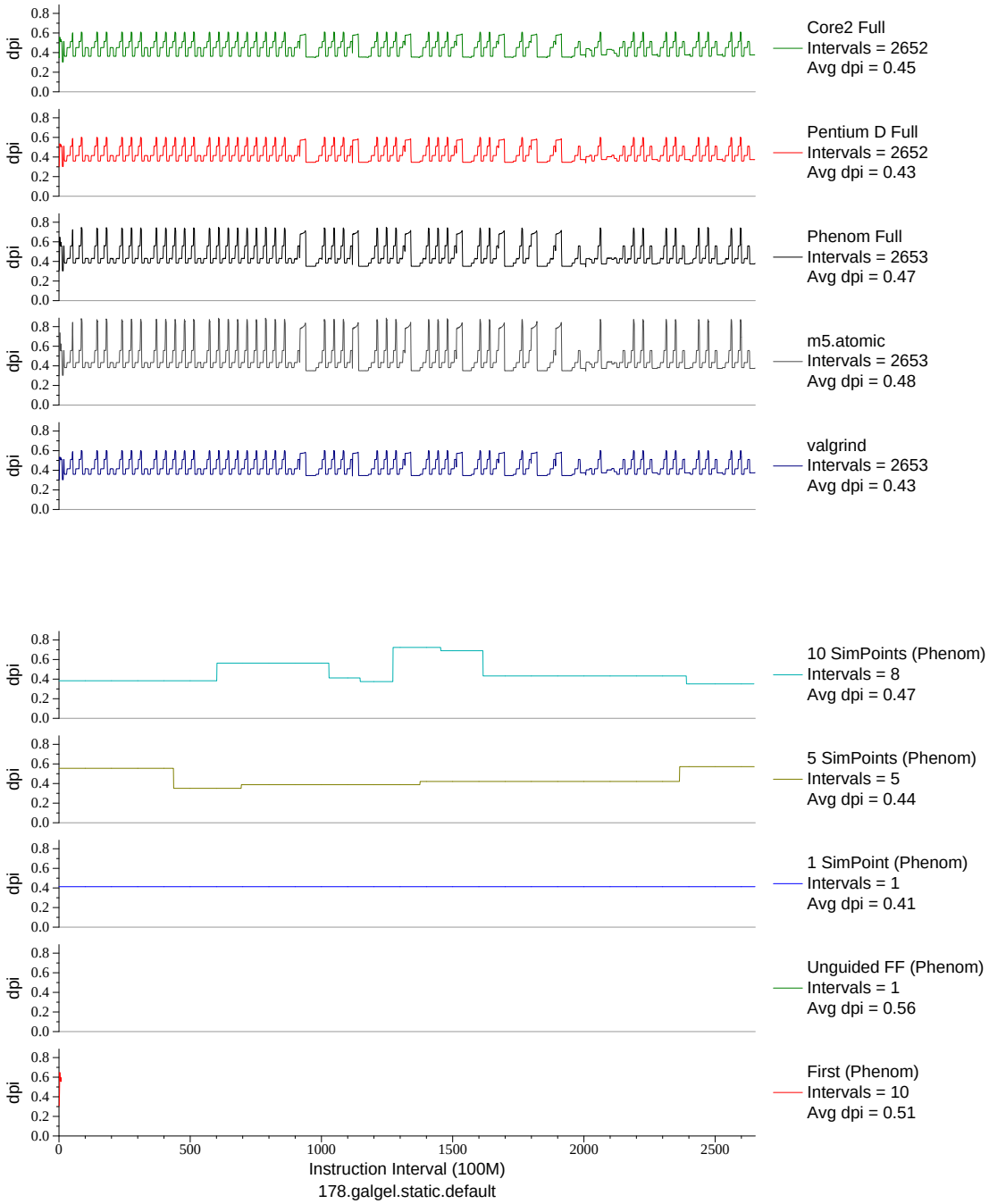

Figure G.18: L1 dcache accesses per instruction plot for `galgel` (FP, F90, Fluid Dynamics)

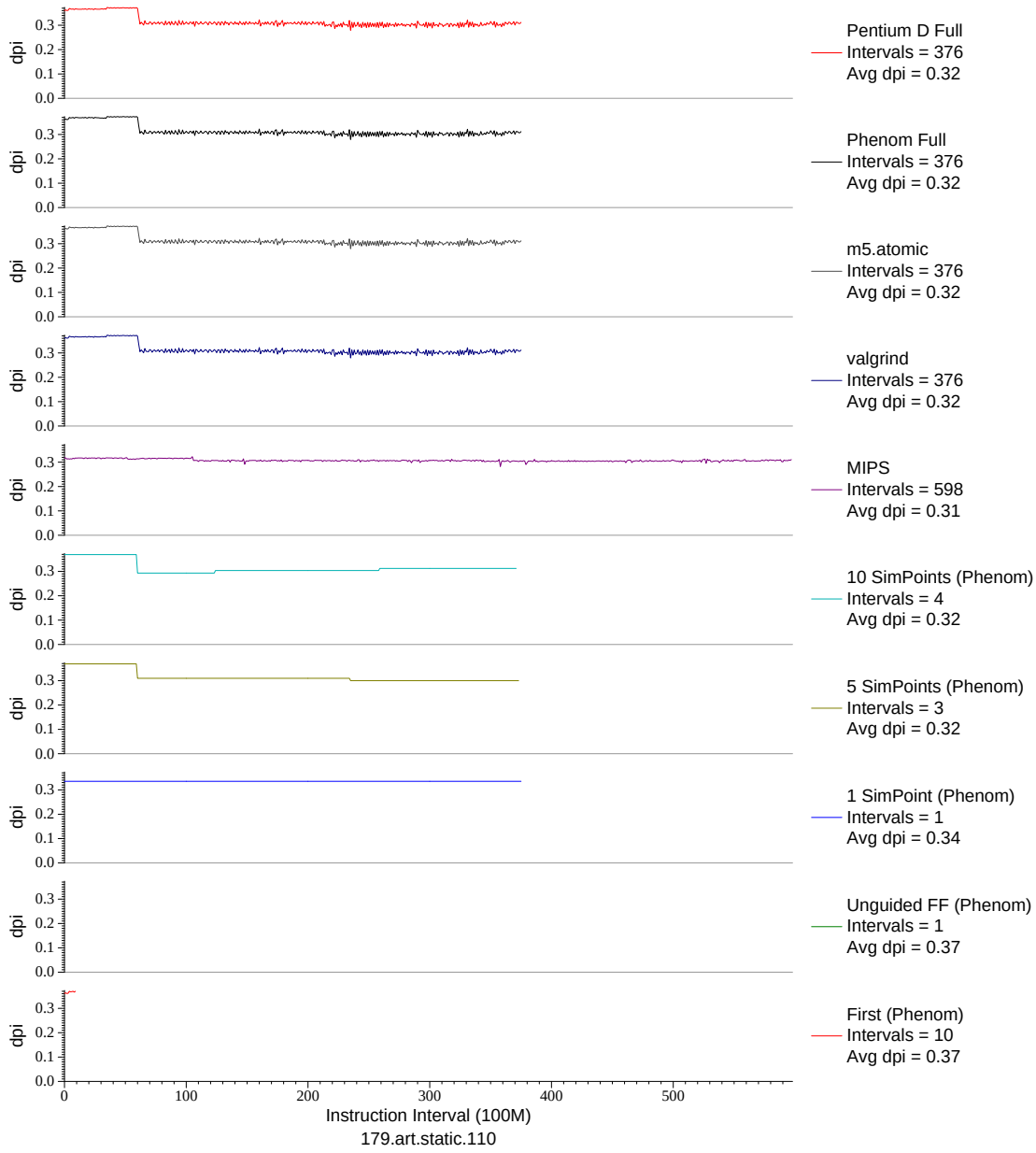

Figure G.19: L1 dcache accesses per instruction plot for `art.110` (FP, C, Neural Networks)

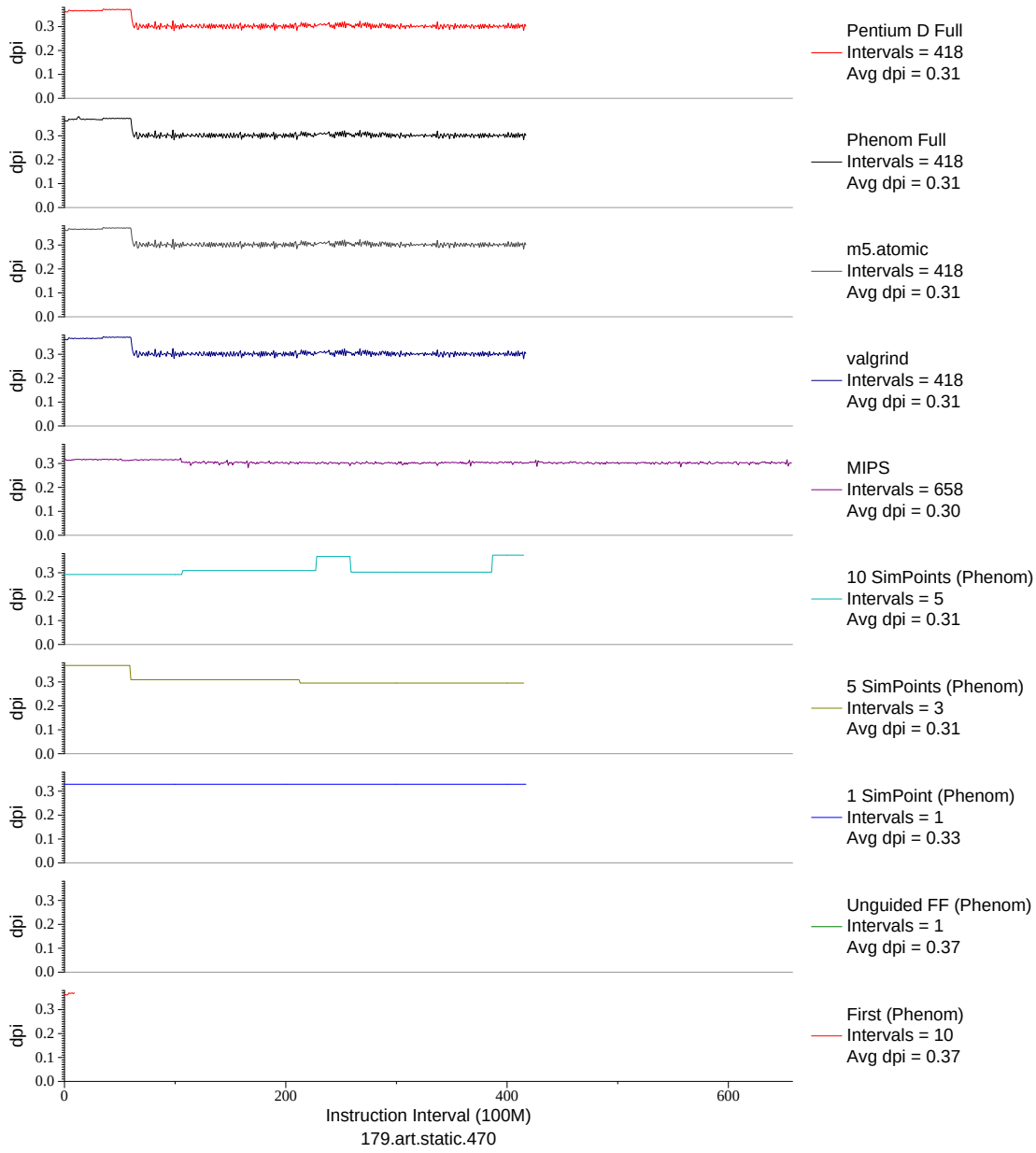

Figure G.20: L1 dcache accesses per instruction plot for `art.470` (FP, C, Neural Networks)

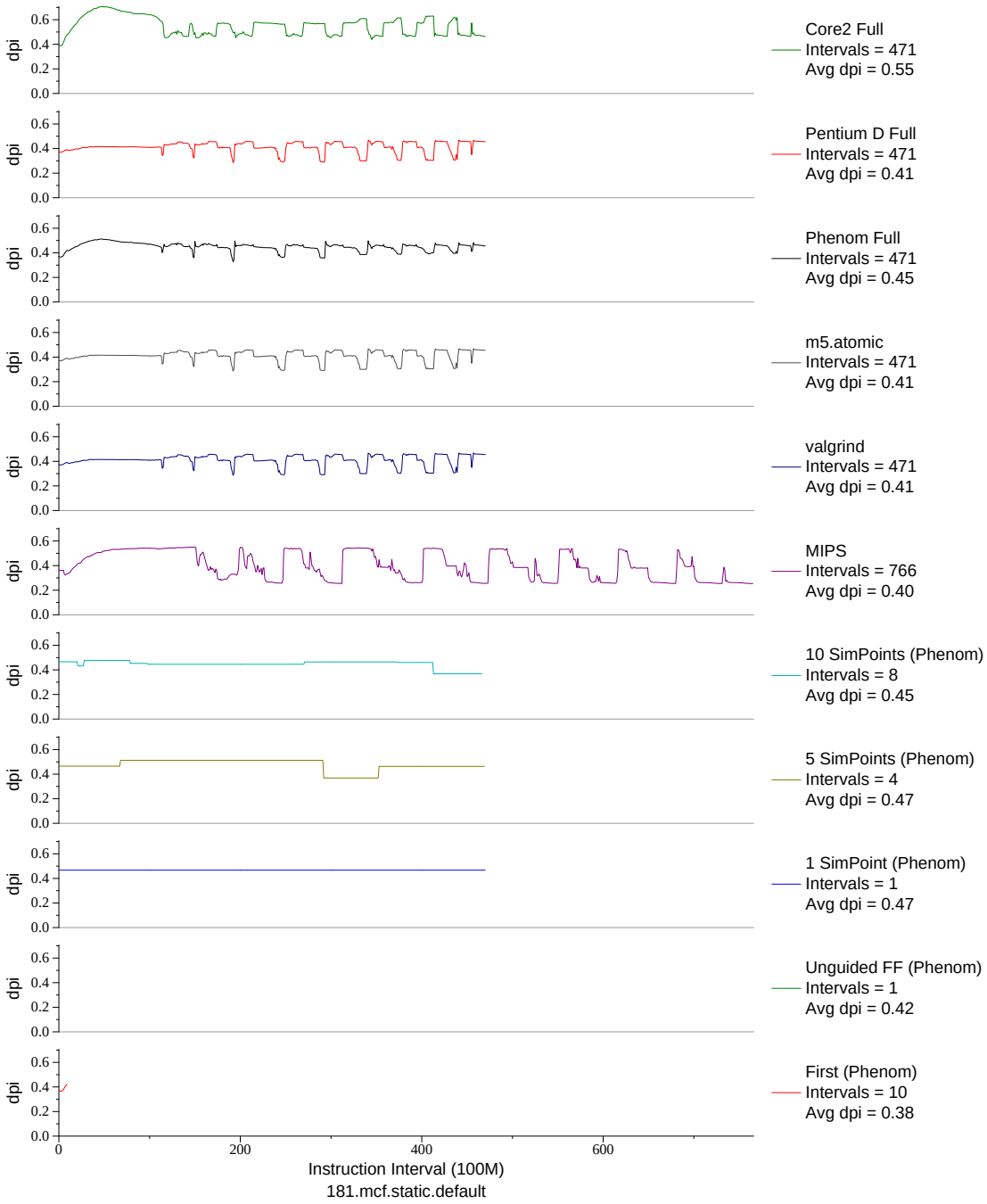

Figure G.21: L1 dcache accesses per instruction plot for `mcf` (INT, C, Combinatorial Opt)

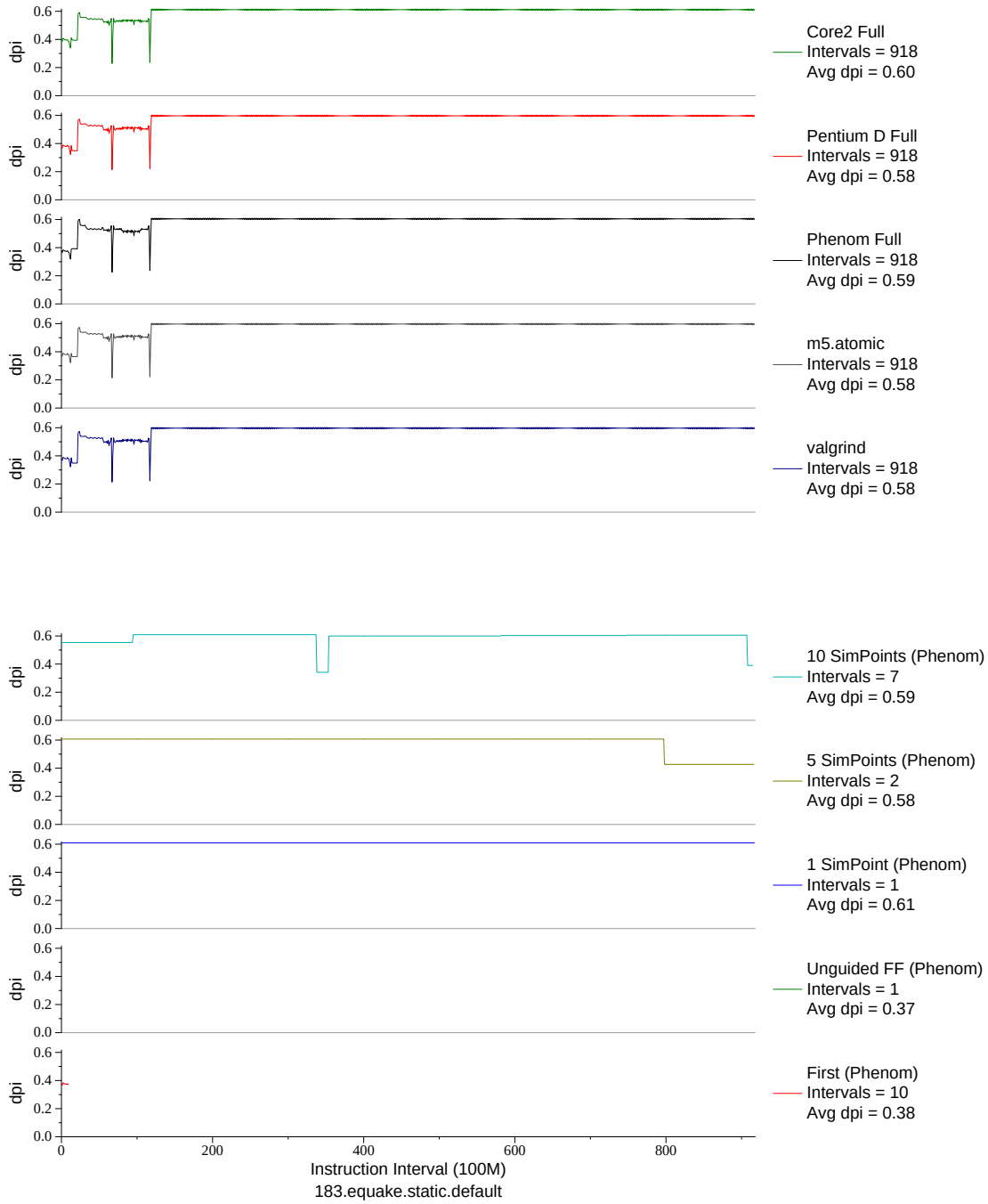

Figure G.22: L1 dcache accesses per instruction plot for `equake` (FP, C, Seismic Propogation)

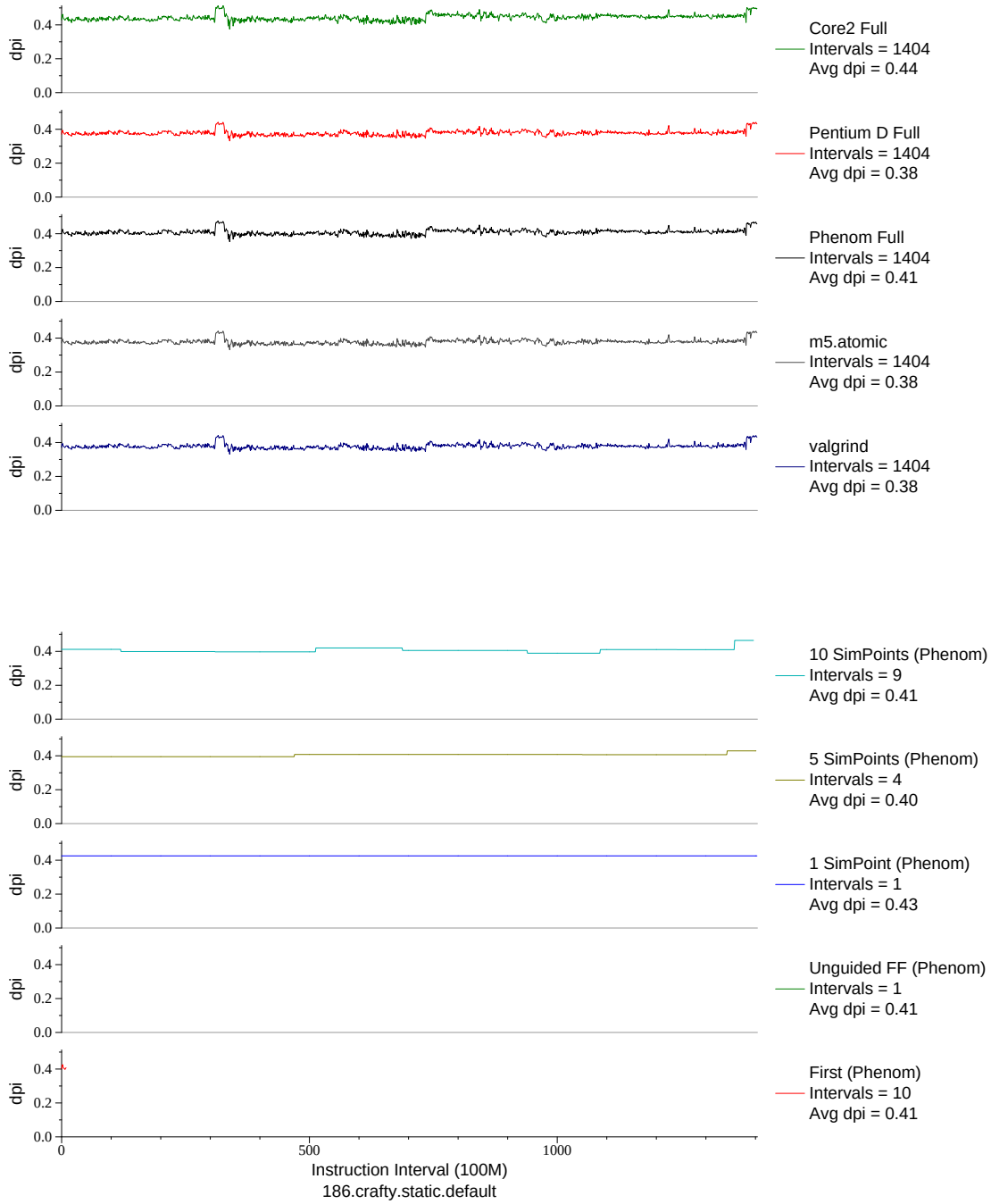

Figure G.23: L1 dcache accesses per instruction plot for `crafty` (INT, C, Chess)

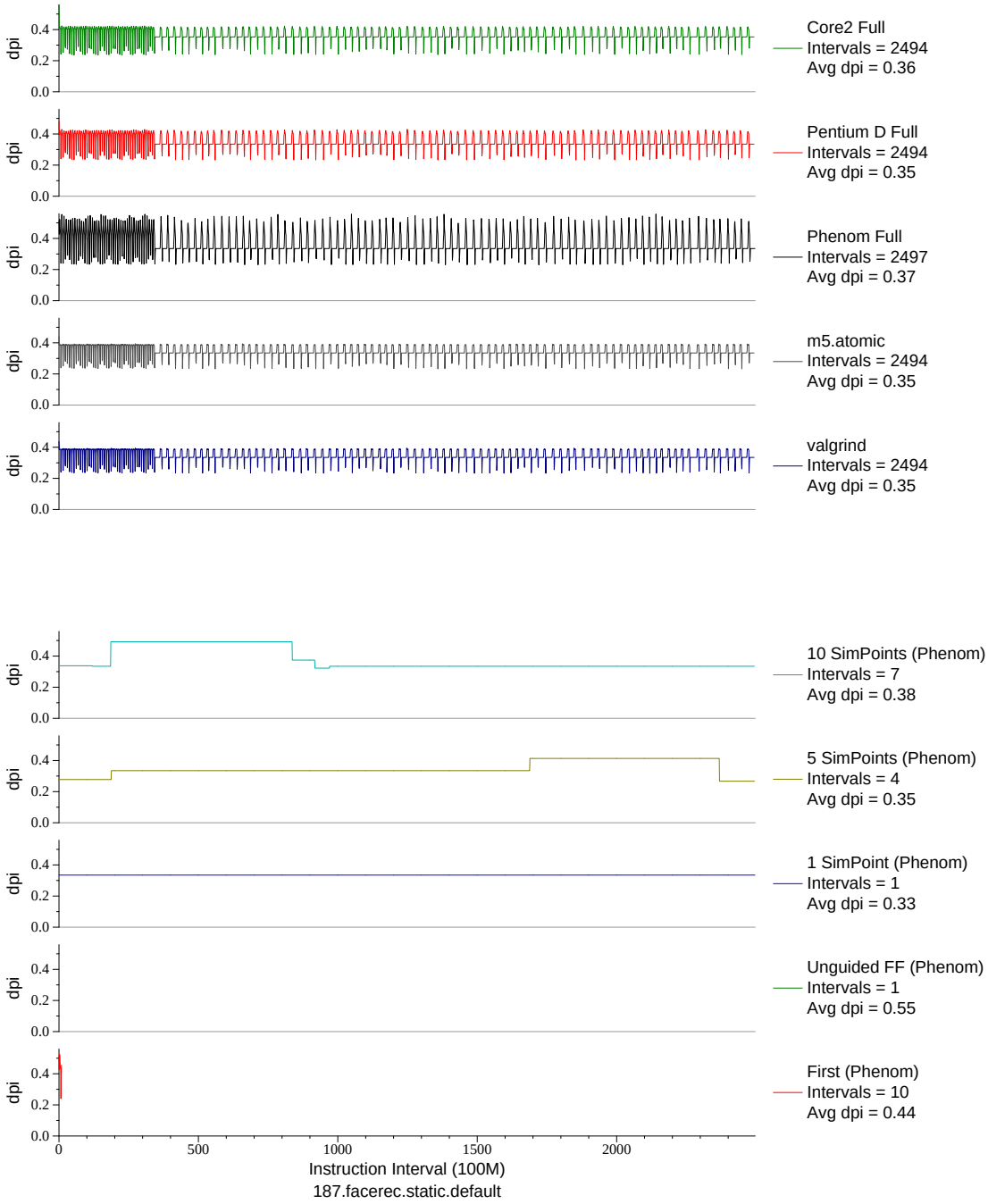

Figure G.24: L1 dcache accesses per instruction plot for `facerec` (FP, F90, Facial Recognition)

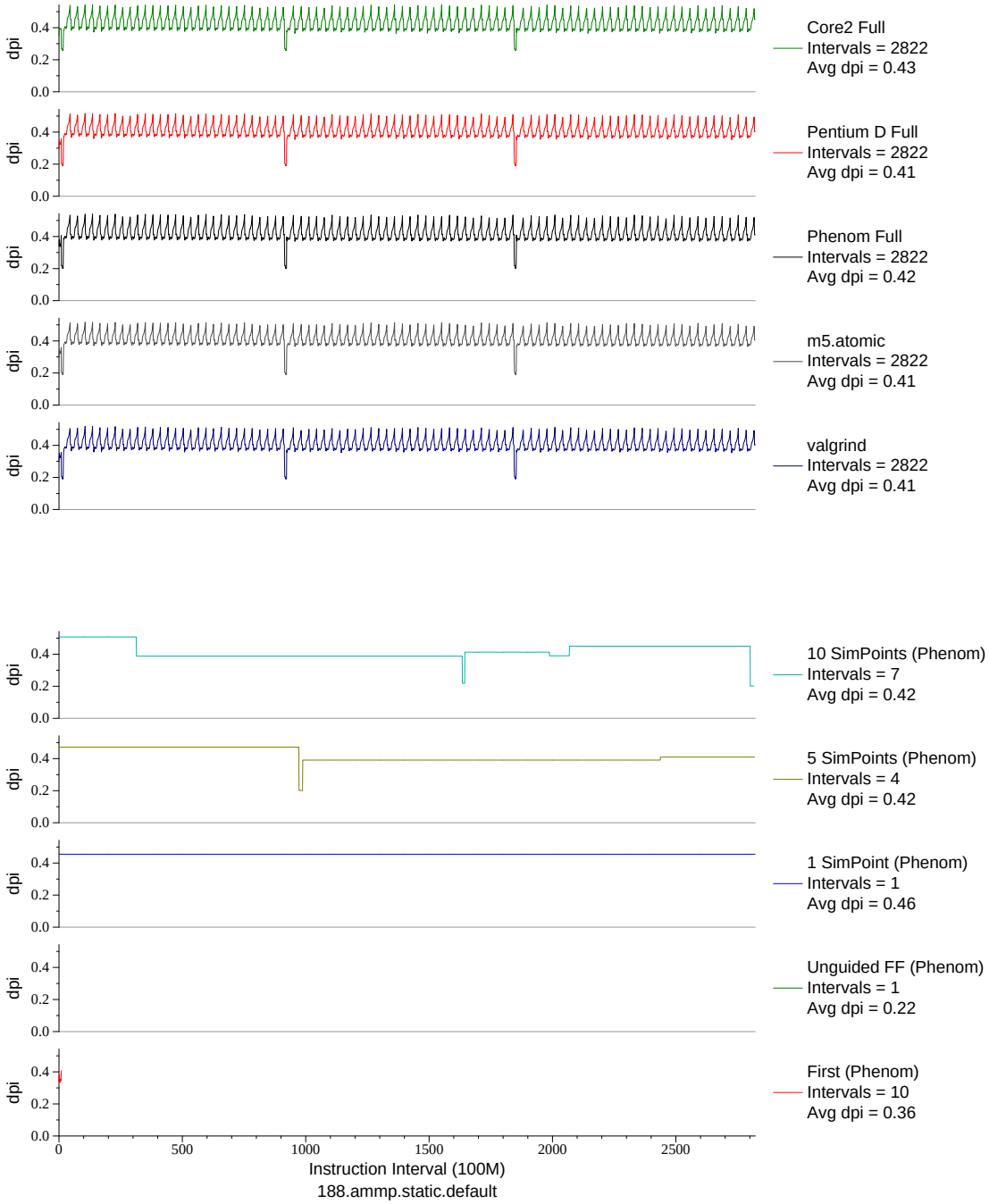

Figure G.25: L1 dcache accesses per instruction plot for `ammp` (FP, C, Chemistry)

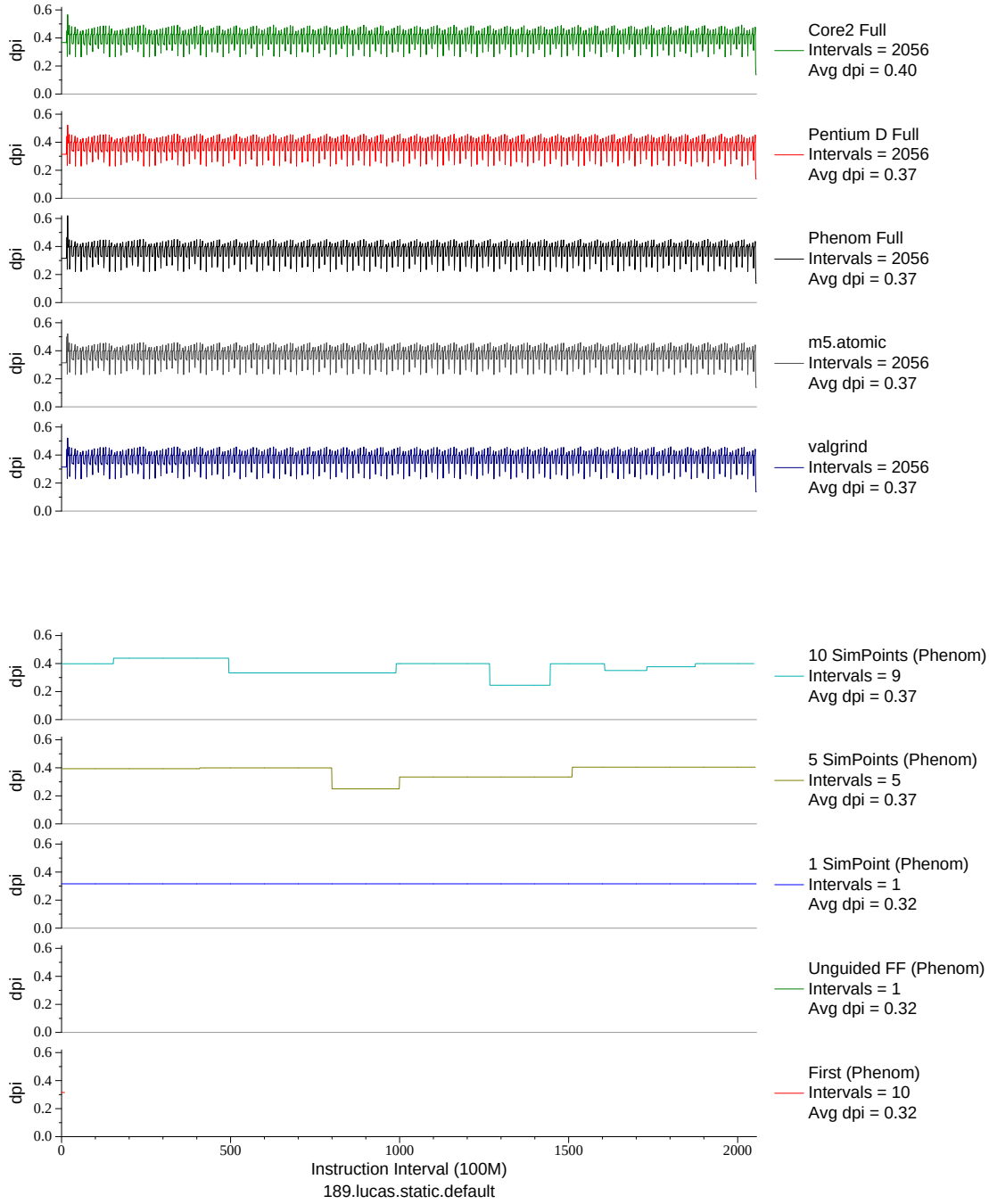

Figure G.26: L1 dcache accesses per instruction plot for `lucas` (FP, F90, Number Theory)

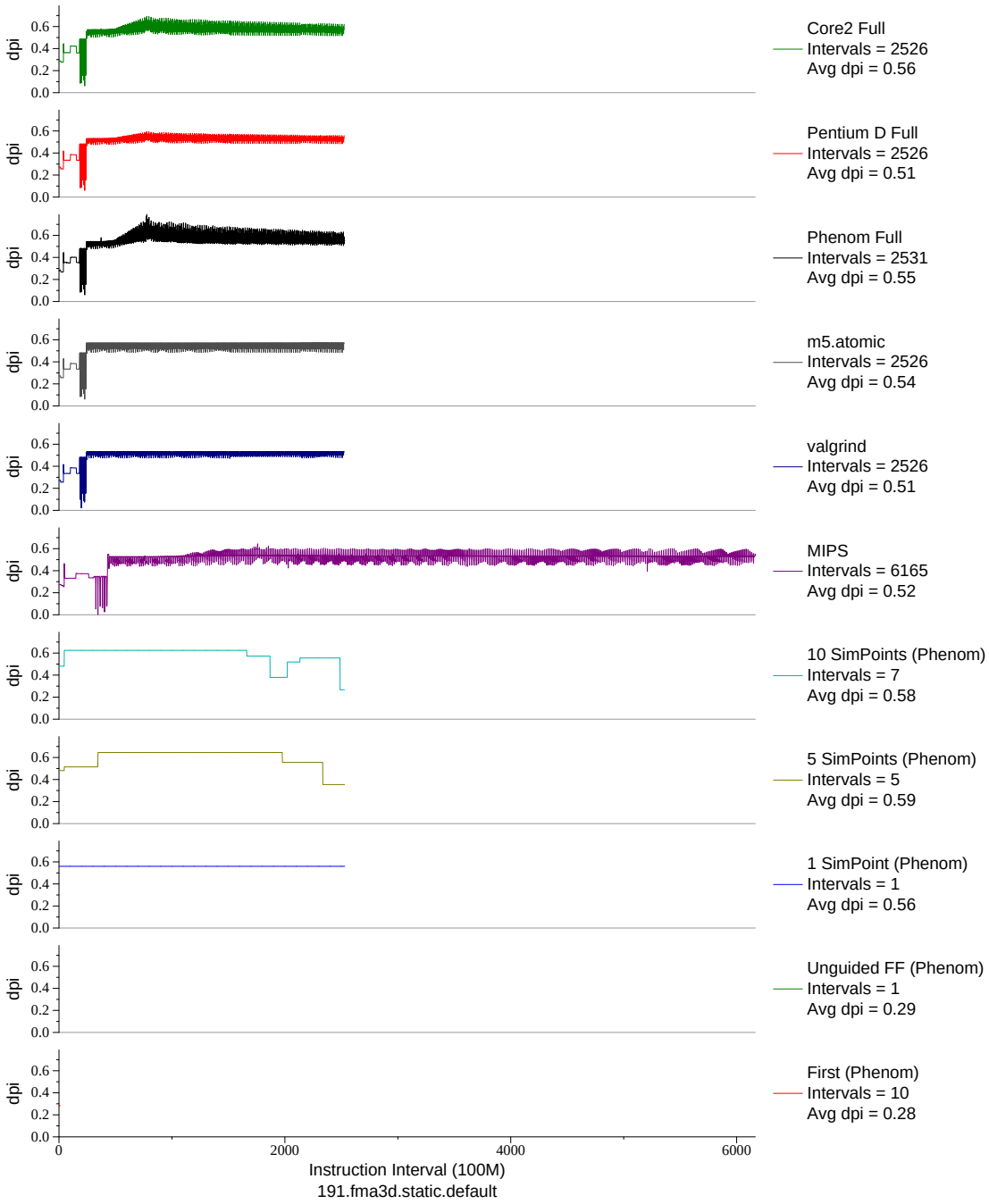

Figure G.27: L1 dcache accesses per instruction plot for `fma3d` (FP, F90, Crash Simulation)

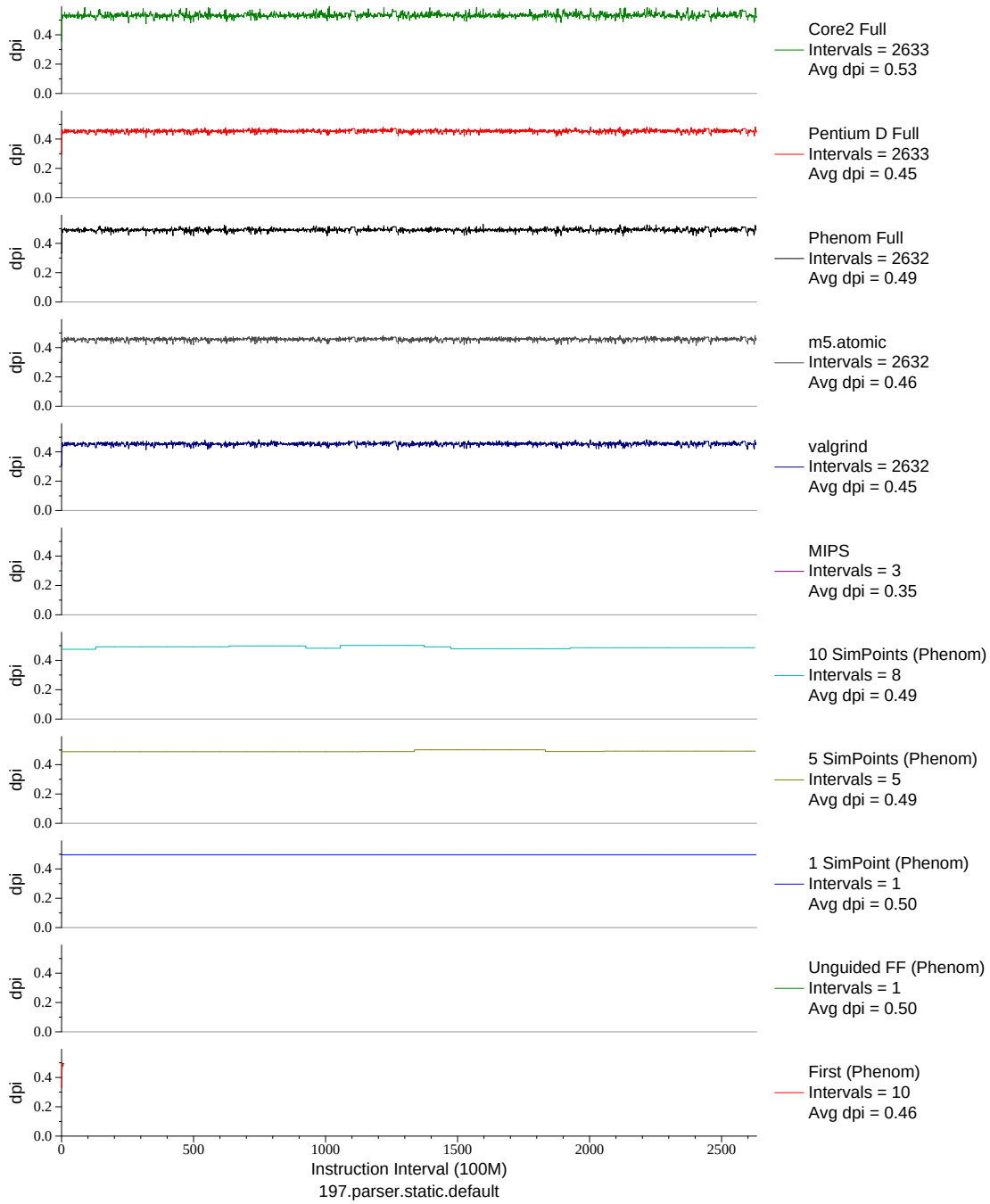

Figure G.28: L1 dcache accesses per instruction plot for `parser` (INT, C, Word Processing)

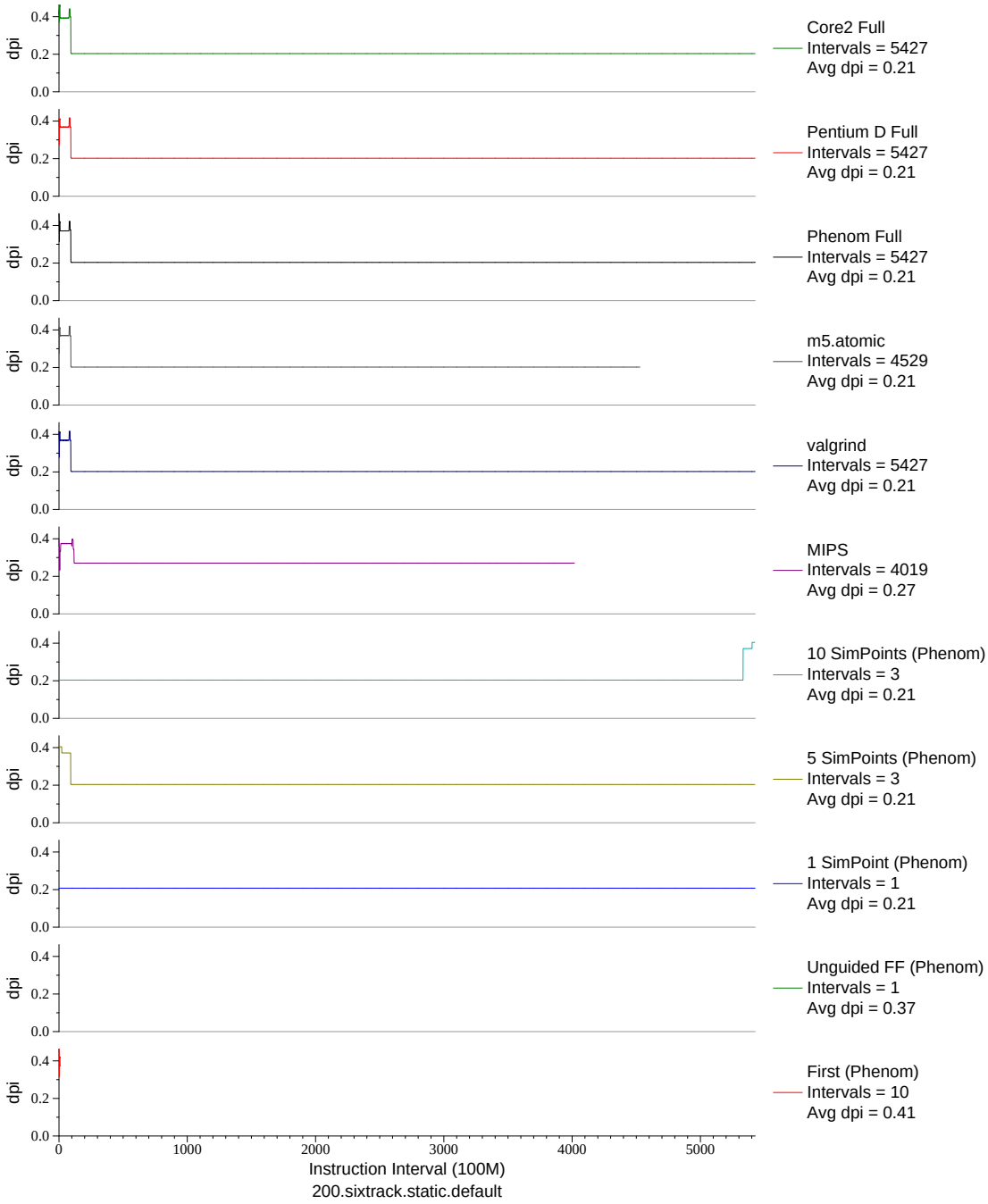

Figure G.29: L1 dcache accesses per instruction plot for `sixtrack` (FP, F77, Nuclear Physics)

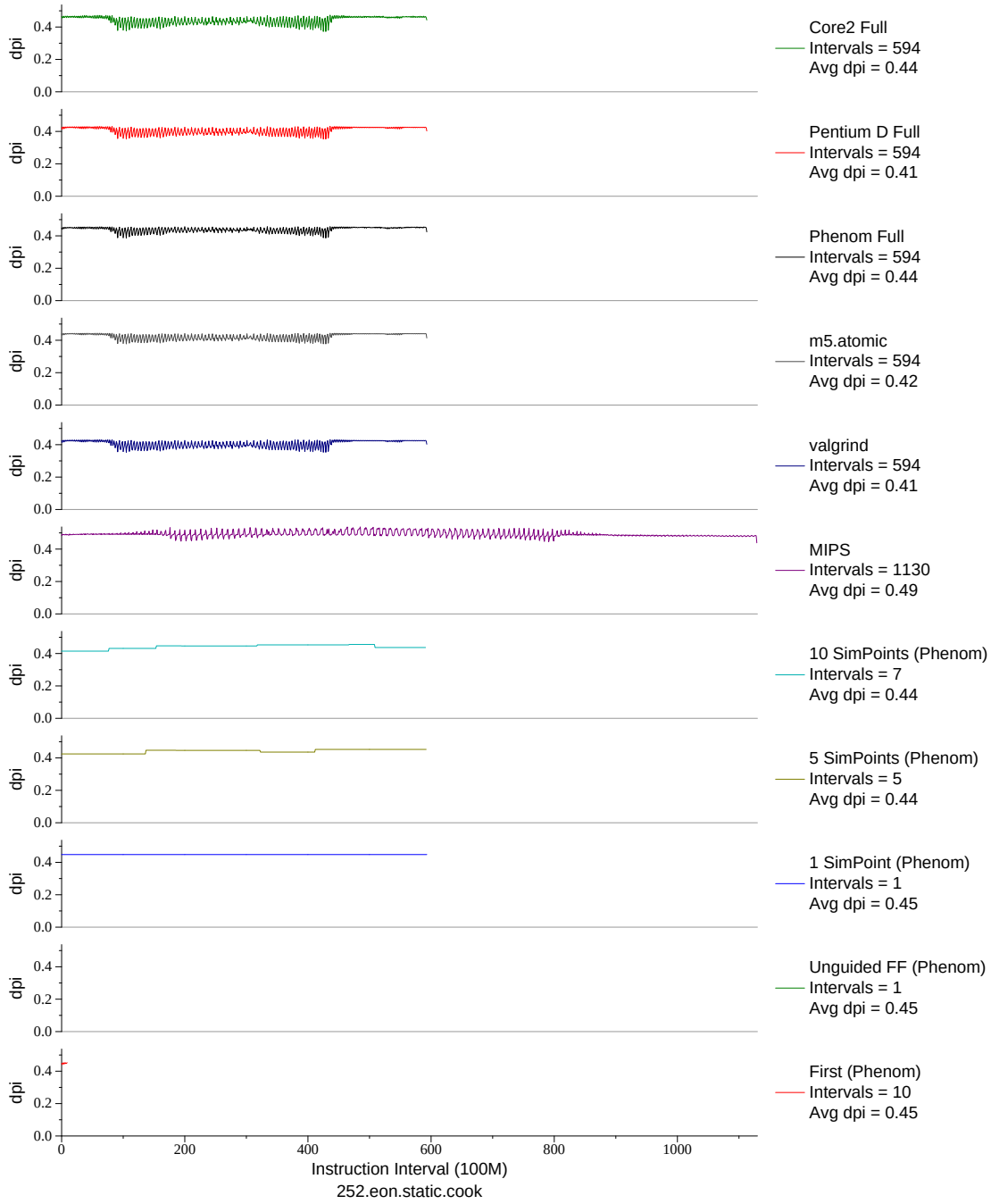

Figure G.30: L1 dcache accesses per instruction plot for `eon.cook` (INT, C++, Computer Graphics)

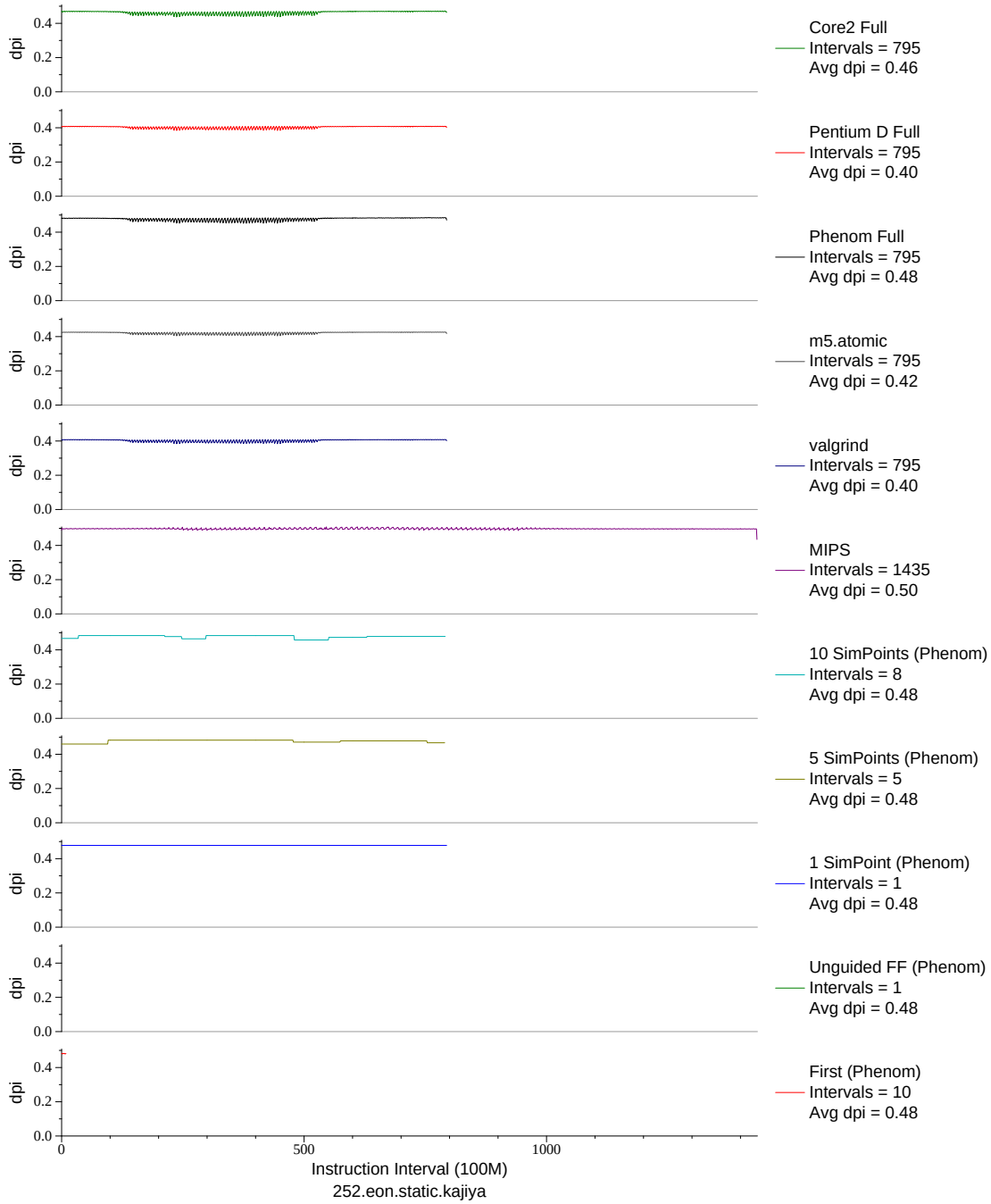

Figure G.31: L1 dcache accesses per instruction plot for `eon.kaj` (INT, C++, Computer Graphics)

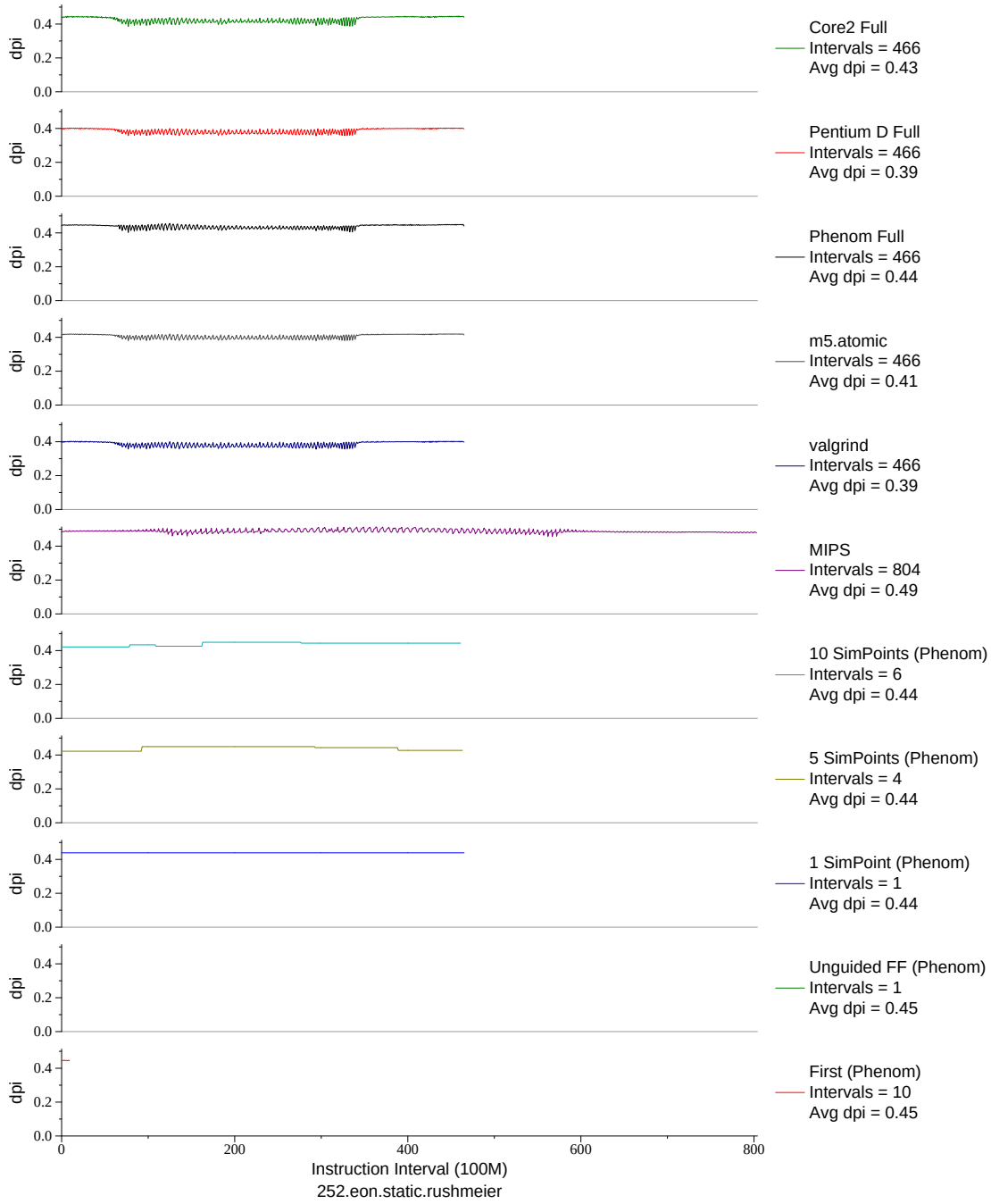

Figure G.32: L1 dcache accesses per instruction plot for `eon.rush` (INT, C++, Computer Graphics)

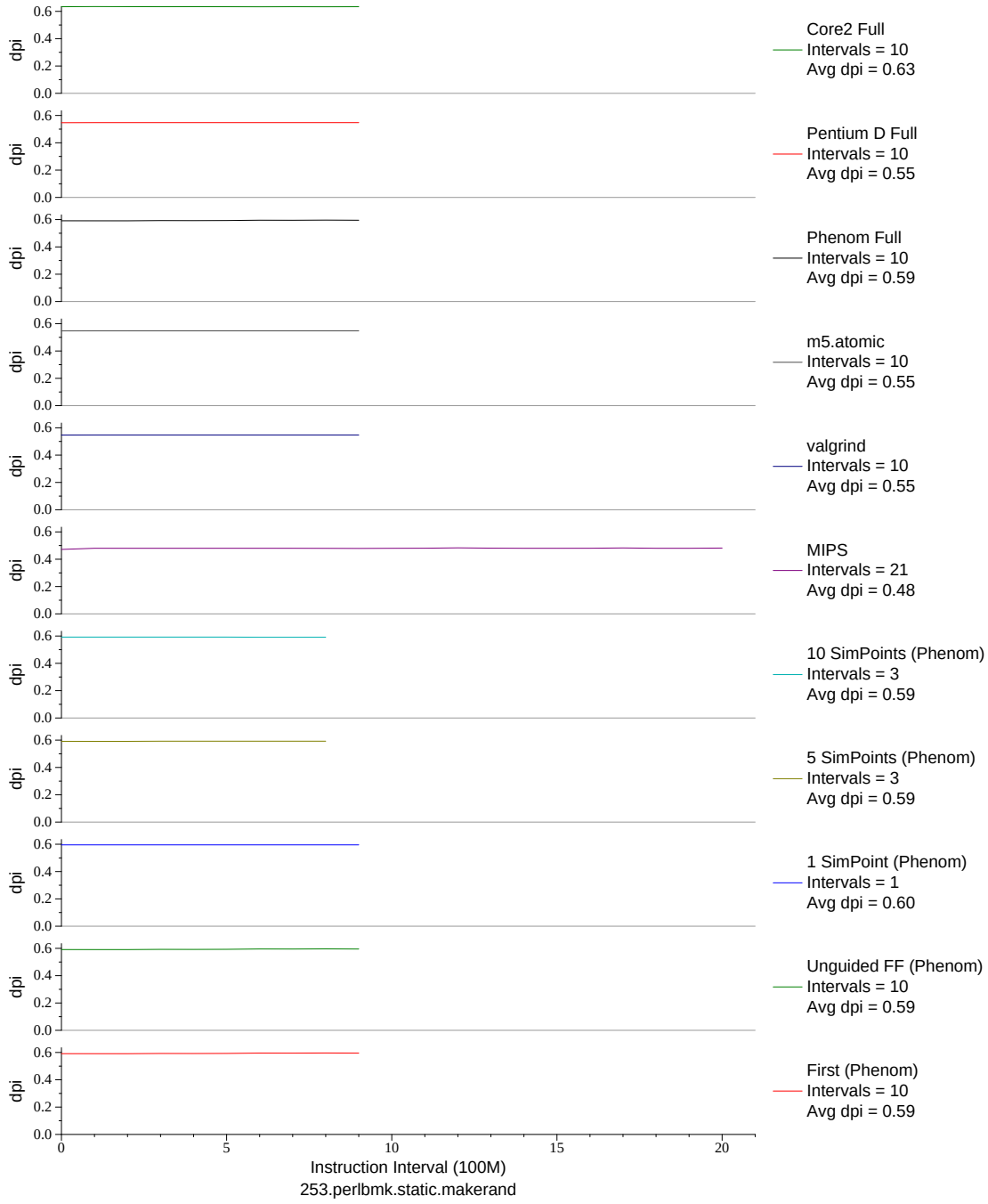

Figure G.33: L1 dcache accesses per instruction plot for `perlbmk.mkrnd` (INT, C, Scripting Language)

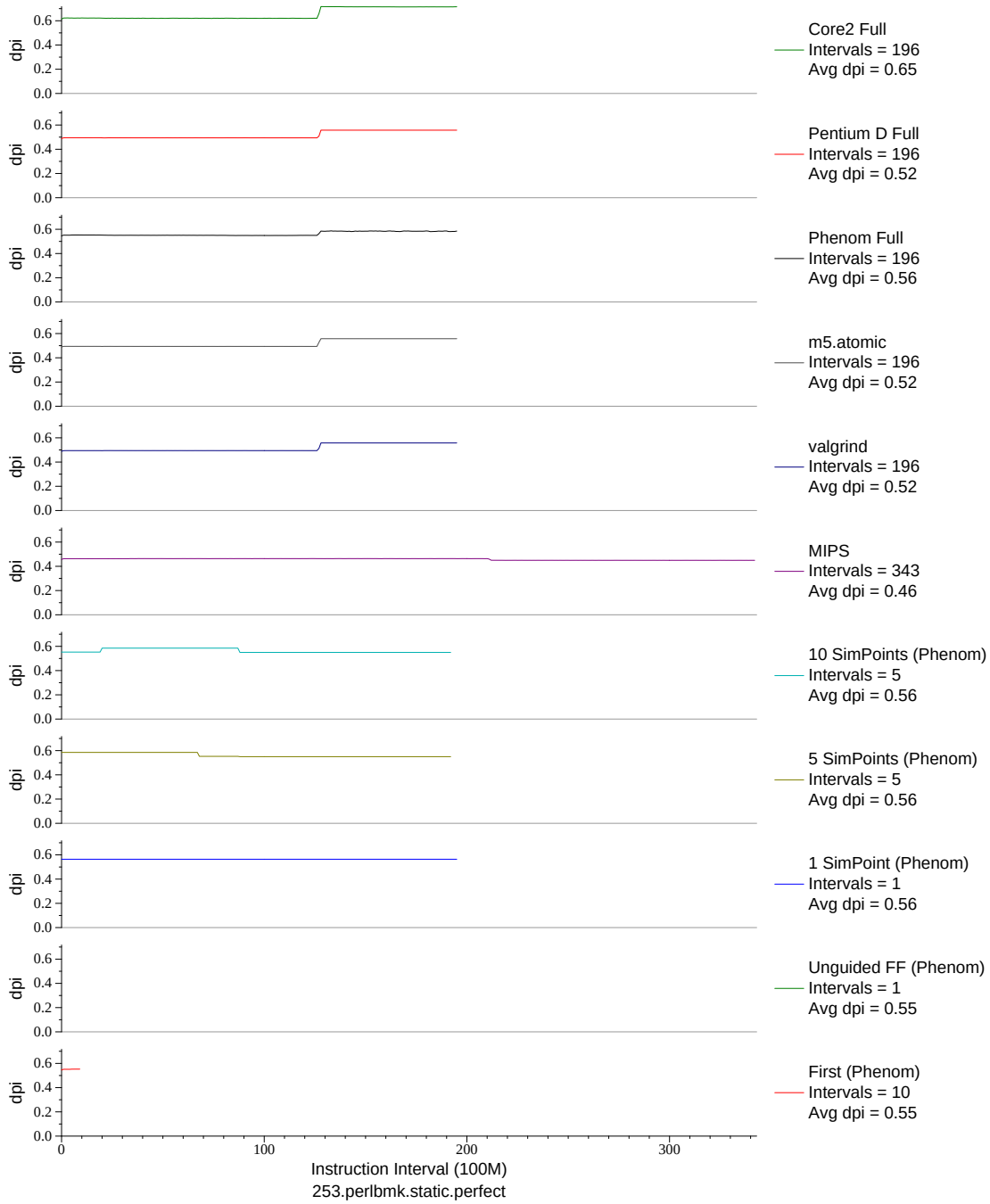

Figure G.34: L1 dcache accesses per instruction plot for `perlbmk.perf` (INT, C, Scripting Language)

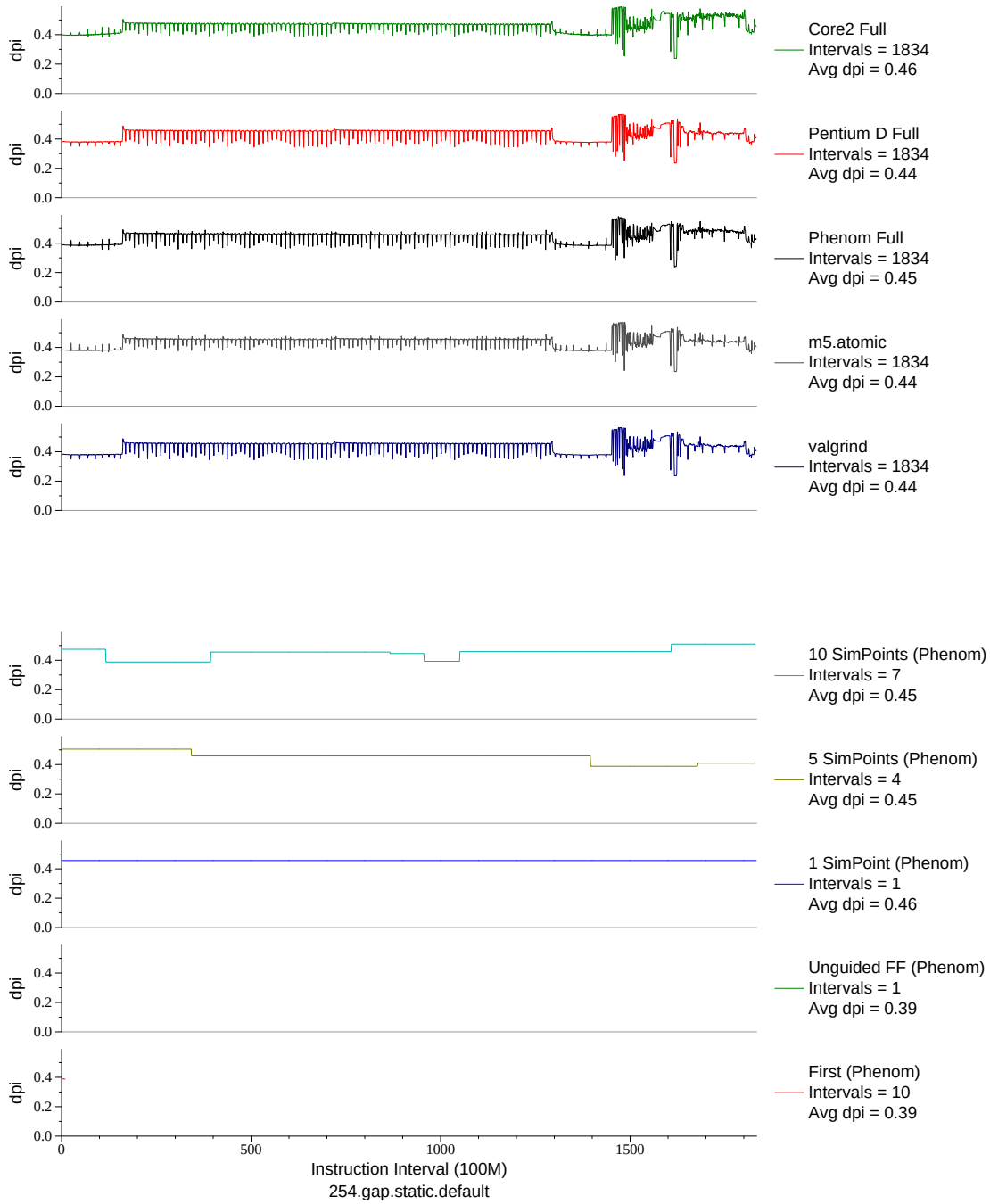

Figure G.35: L1 dcache accesses per instruction plot for `gap` (INT, C, Group Theory)

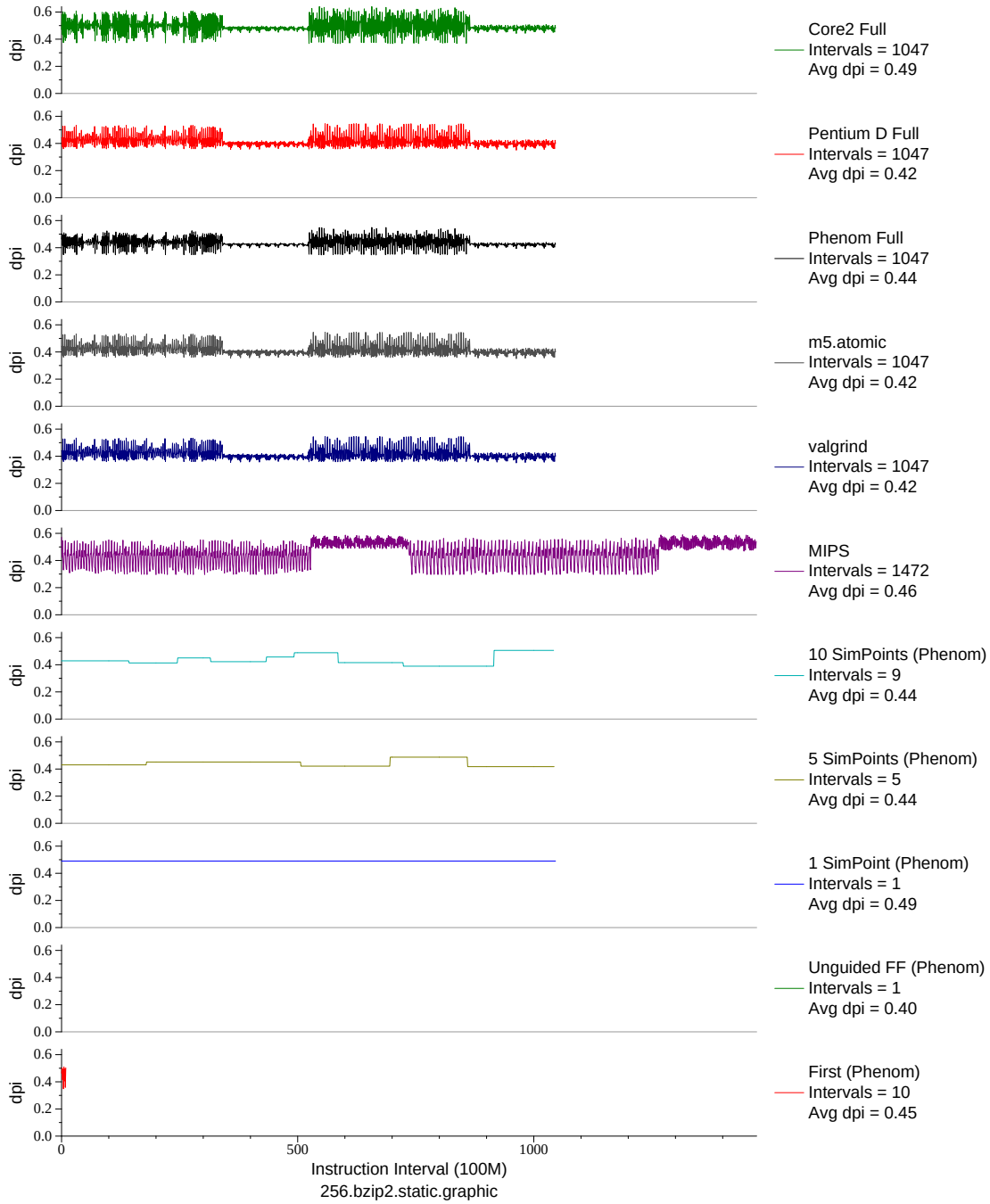

Figure G.36: L1 dcache accesses per instruction plot for `bzip2.graph` (INT, C, Compression)

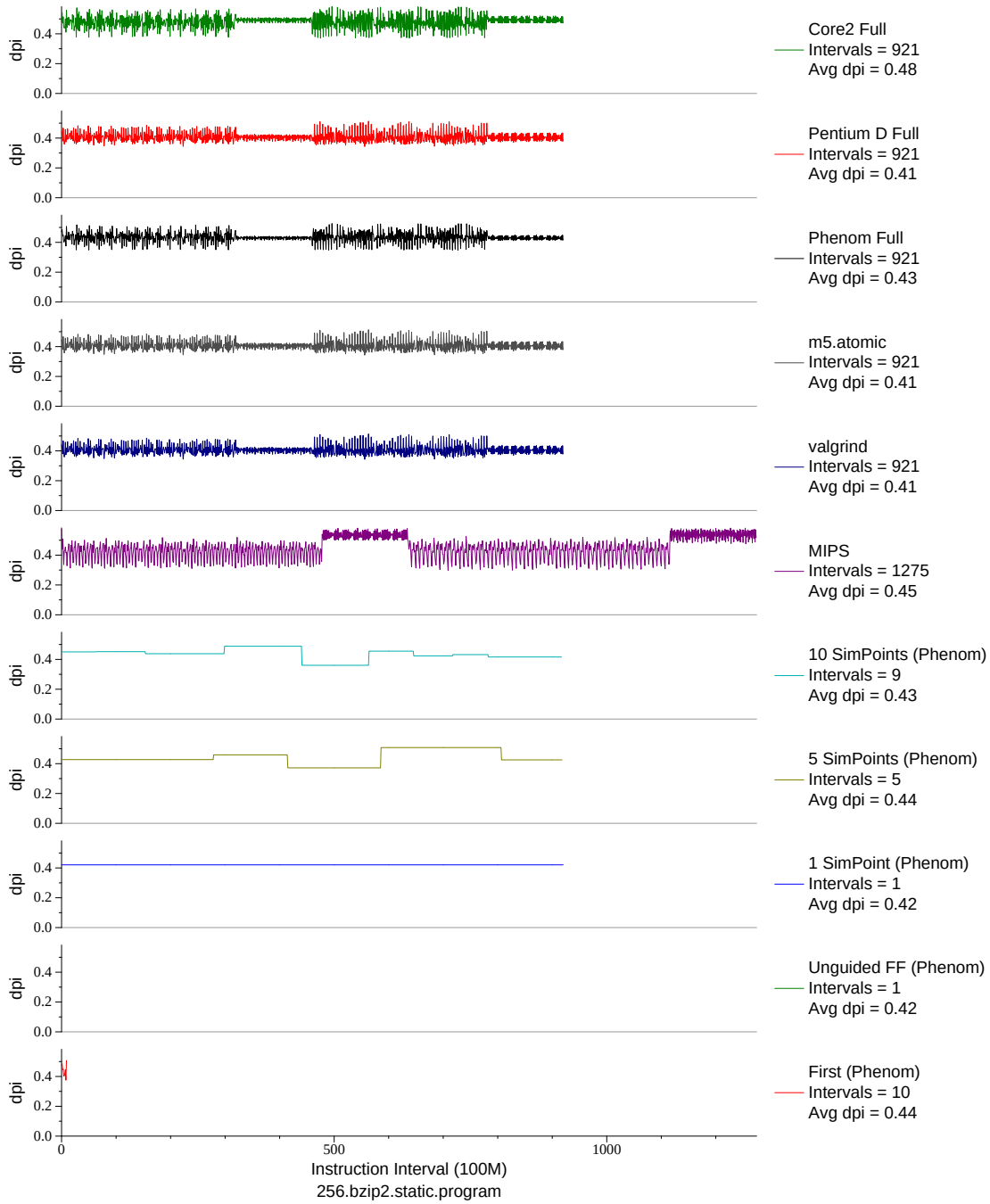

Figure G.37: L1 dcache accesses per instruction plot for `bzip2.prog` (INT, C, Compression)

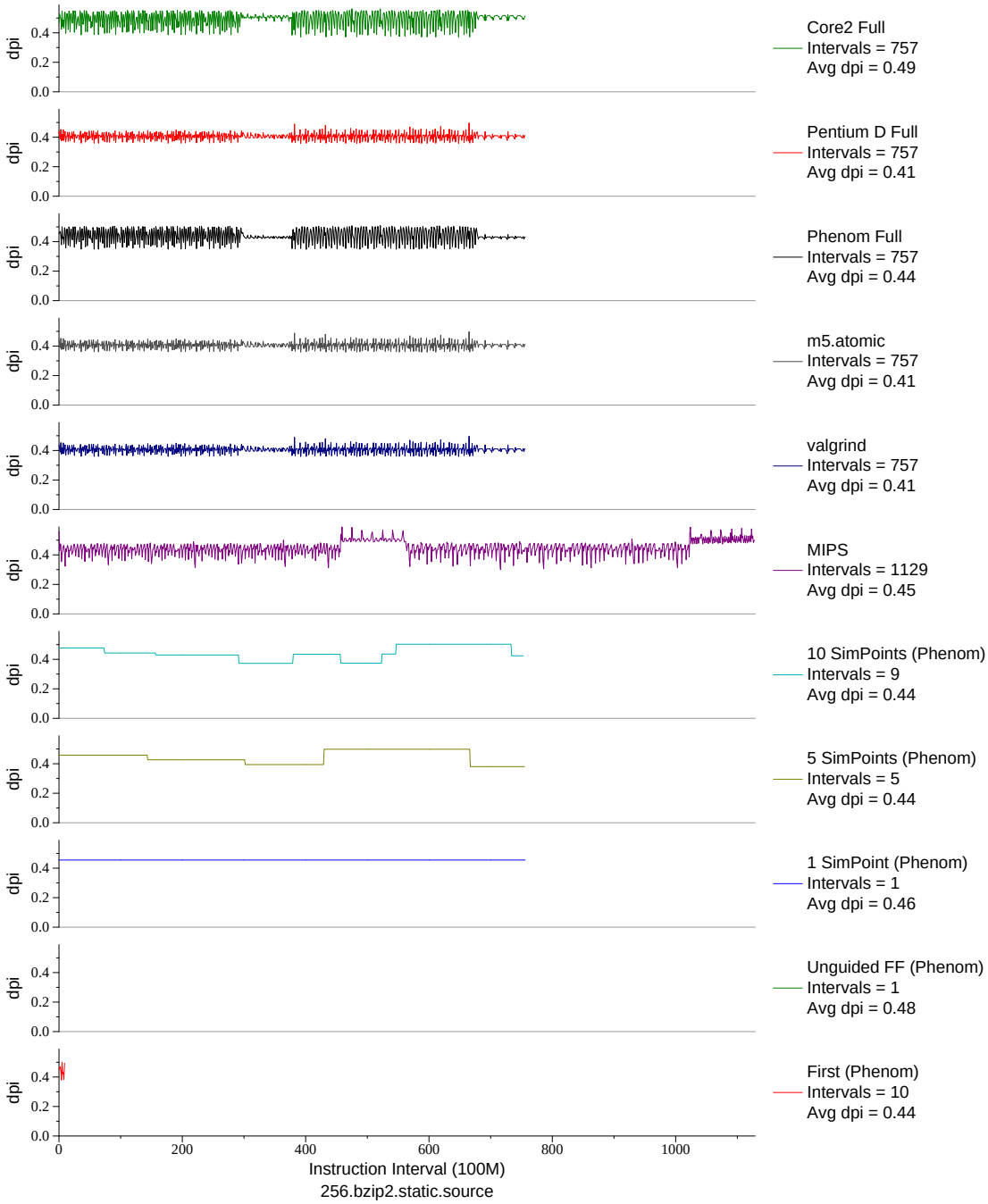

Figure G.38: L1 dcache accesses per instruction plot for `bzip2.src` (INT, C, Compression)

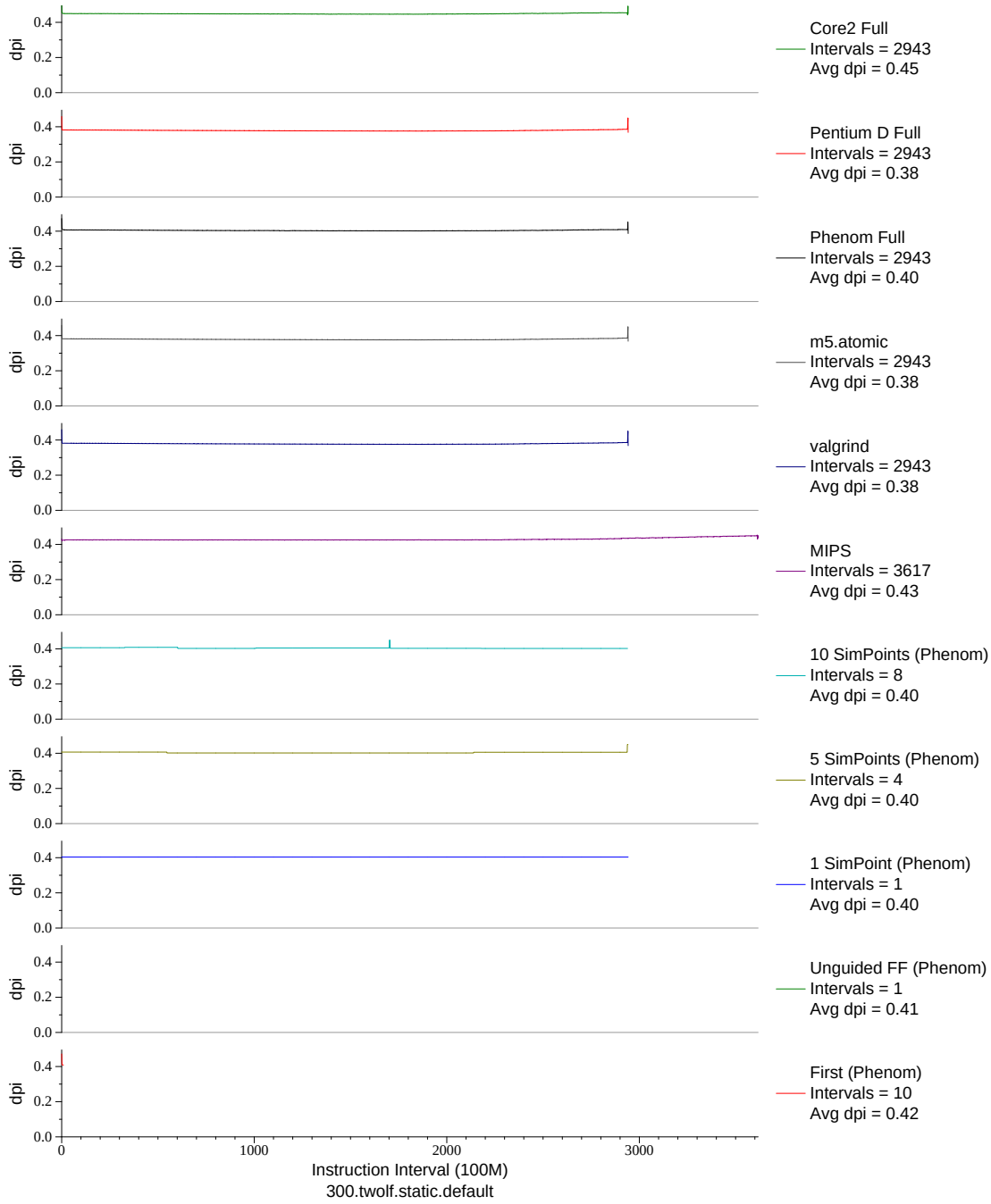

Figure G.39: L1 dcache accesses per instruction plot for `twolf` (INT, C, Place/Route)

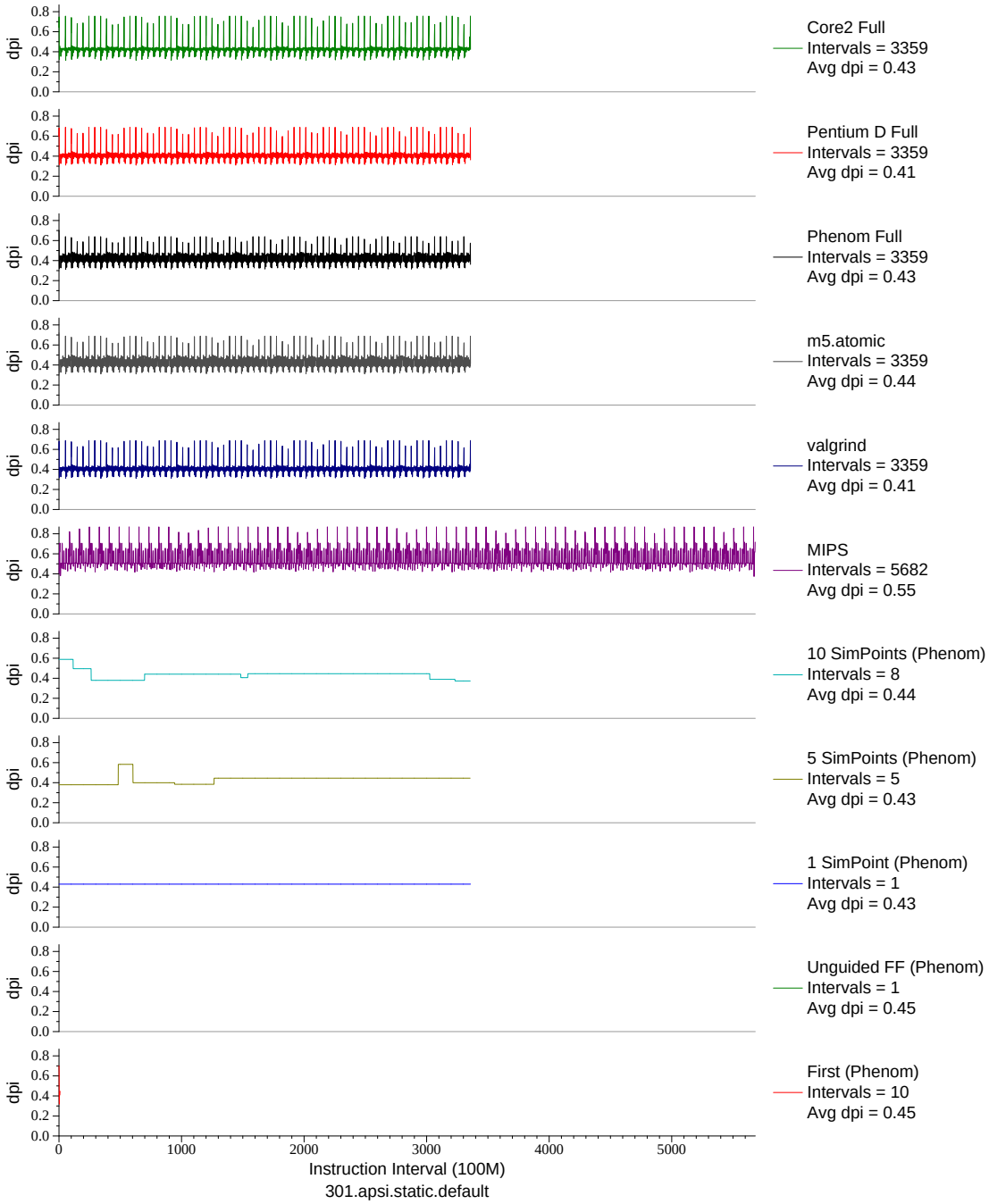

Figure G.40: L1 dcache accesses per instruction plot for `apsi` (FP, F77, Meteorology/Pollution)

APPENDIX H

L1 DATA CACHE ACCESSES PER μOP PHASE PLOTS

On the x86 architecture instructions are decoded into RISC-like μops before execution. Each implementation of the architecture has a different set of μops, making comparisons difficult. What follows are phase plots showing data cache accesses per μop for SPEC CPU2000 on three different x86_64 machines, as well as MIPS (to show how RISC-like the μops are).

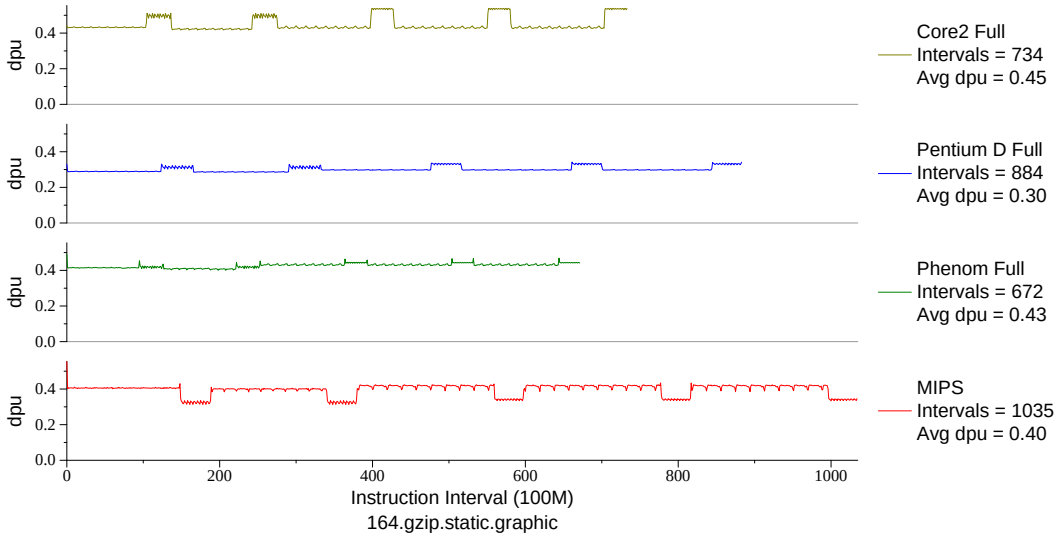

Figure H.1: L1 D$ accesses per μop for `gzip.graph` (INT, C, Compression)

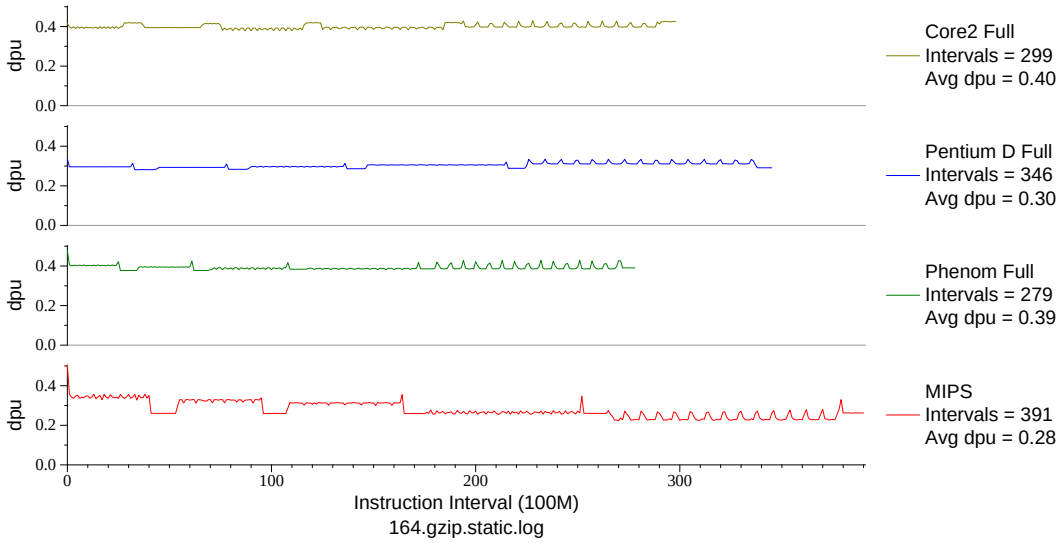

Figure H.2: L1 D$ accesses per μop for `gzip.log` (INT, C, Compression)

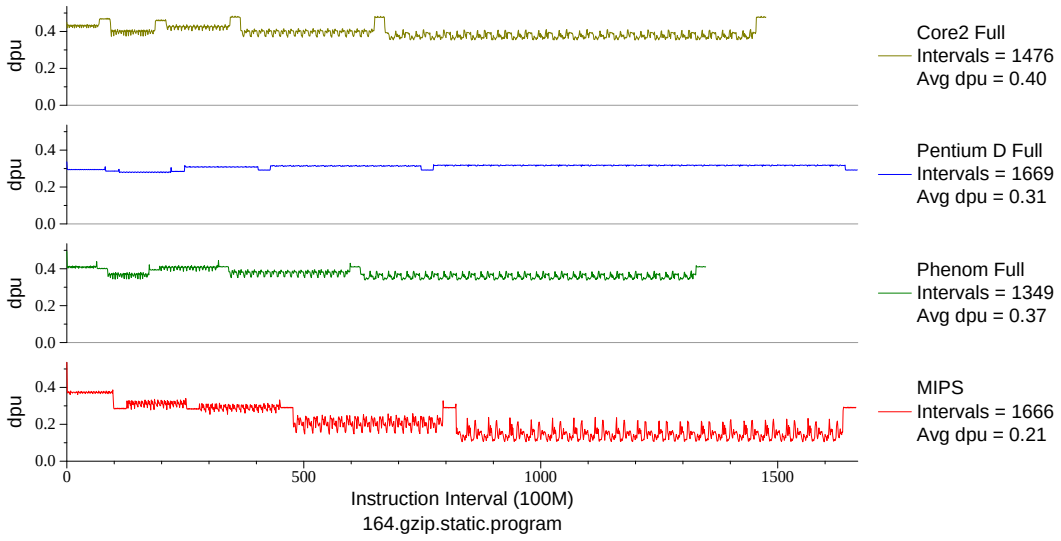

Figure H.3: L1 D$ accesses per μop for `gzip.prog` (INT, C, Compression)

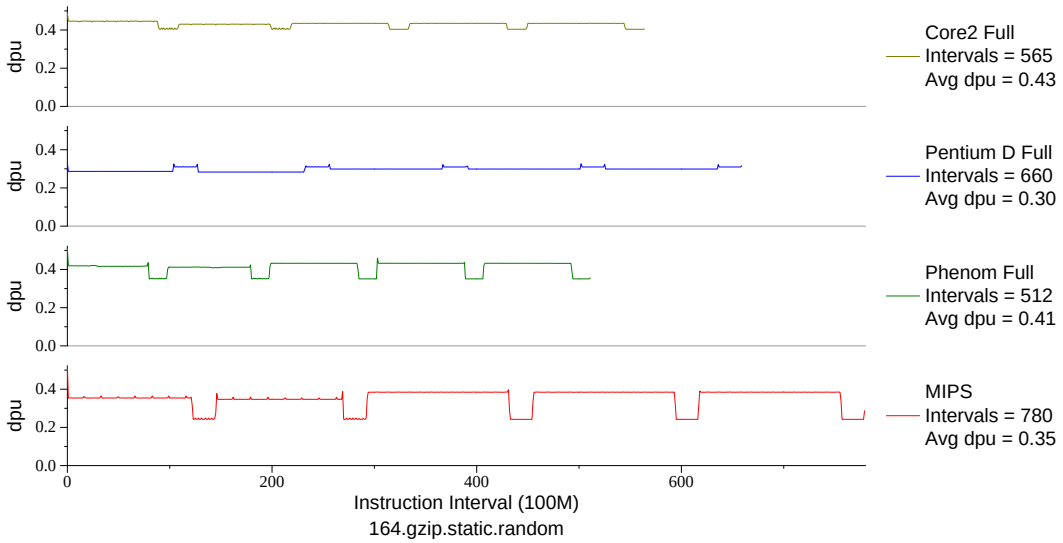

Figure H.4: L1 D$ accesses per μop for `gzip.rand` (INT, C, Compression)

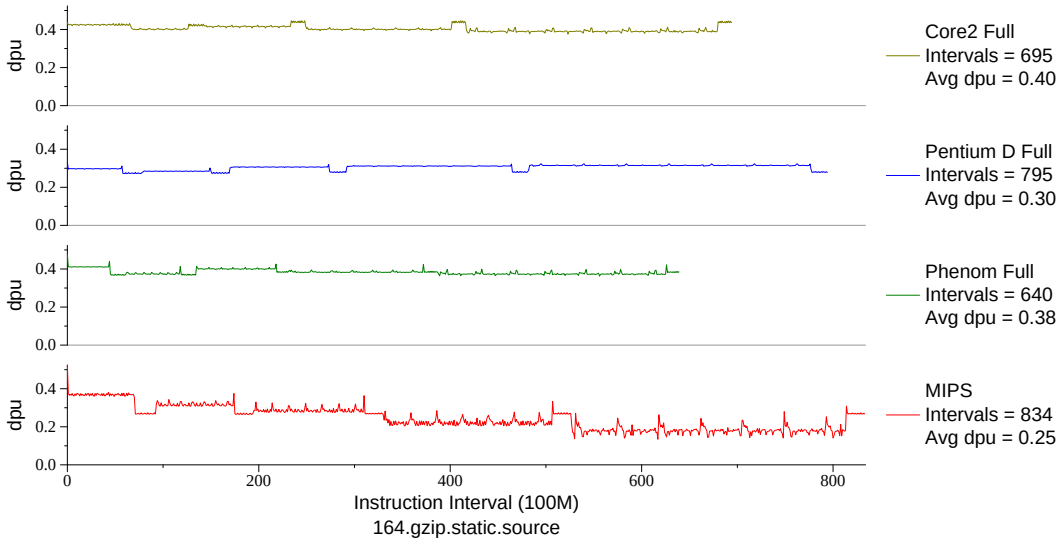

Figure H.5: L1 D$ accesses per μop for `gzip.src` (INT, C, Compression)

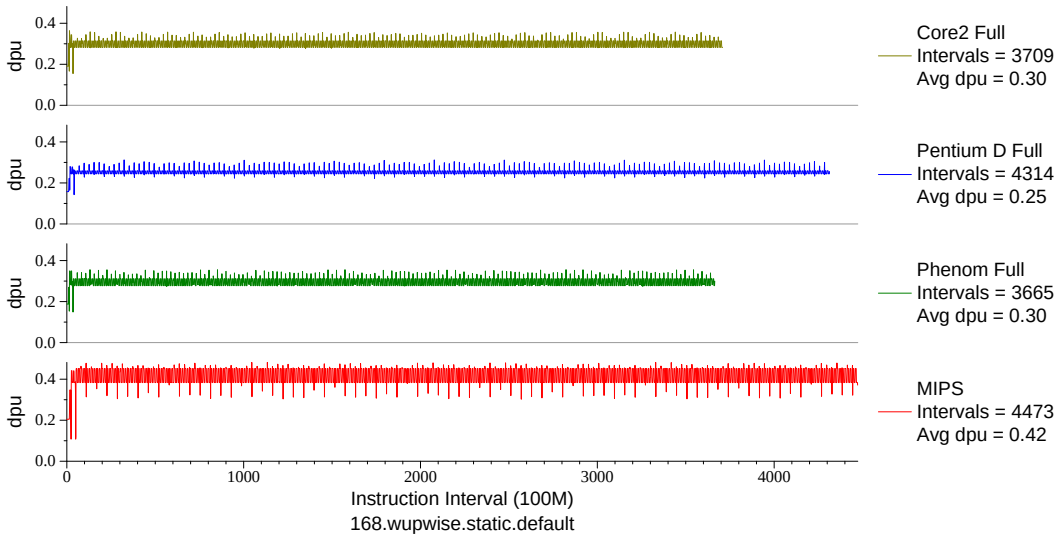

Figure H.6: L1 D$ accesses per μop for `wupwise` (FP, F77, Quantum Chromodynamics)

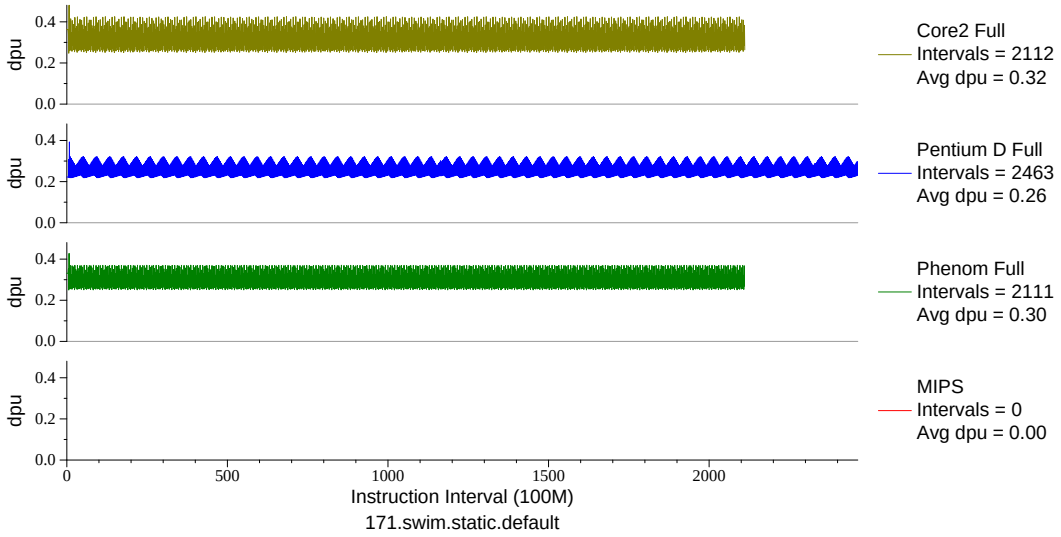

Figure H.7: L1 D$ accesses per μop for `swim` (FP, F77, Meteorology/Water)

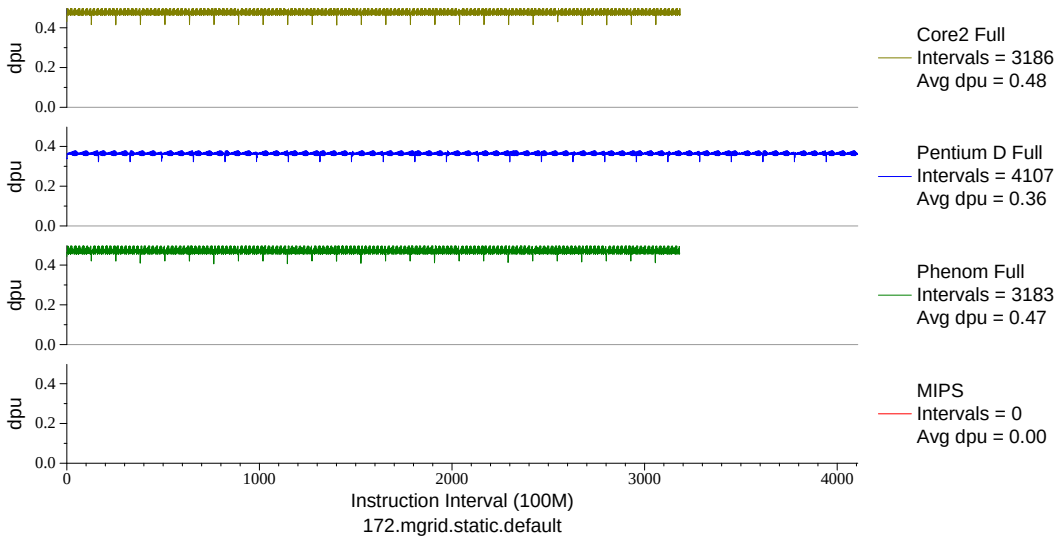

Figure H.8: L1 D$ accesses per μop for `mgrid` (FP, F77, Multi-Grid Solver)

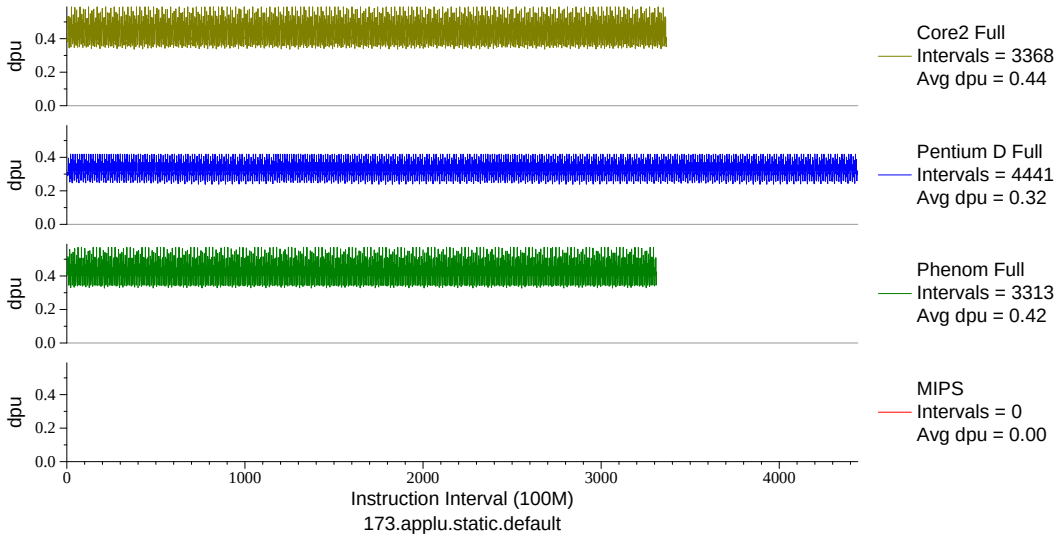

Figure H.9: L1 D$ accesses per μop for `applu` (FP, F77, Fluid Dynamics)

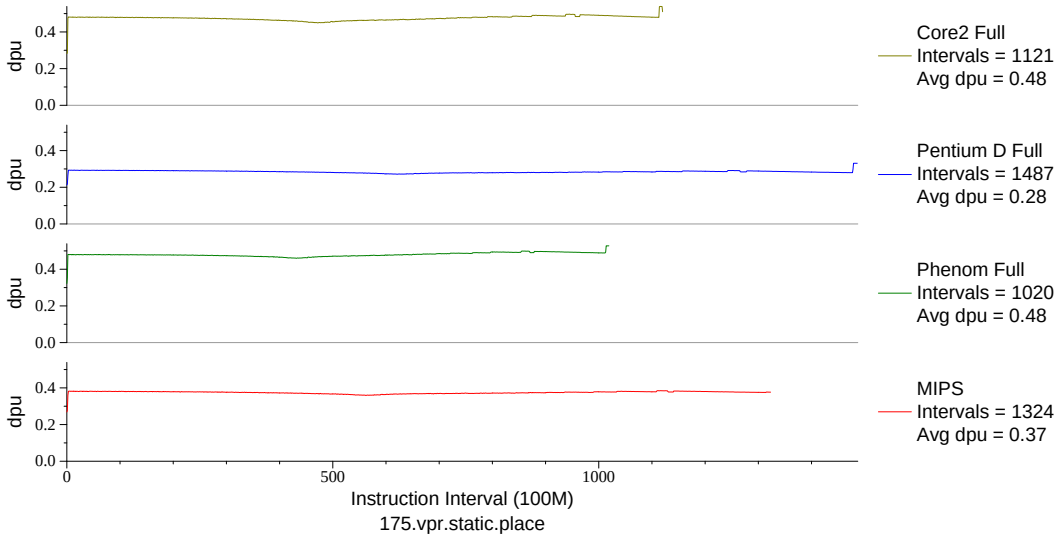

Figure H.10: L1 D$ accesses per μop for `vpr.place` (INT, C, FPGA Place/Route)

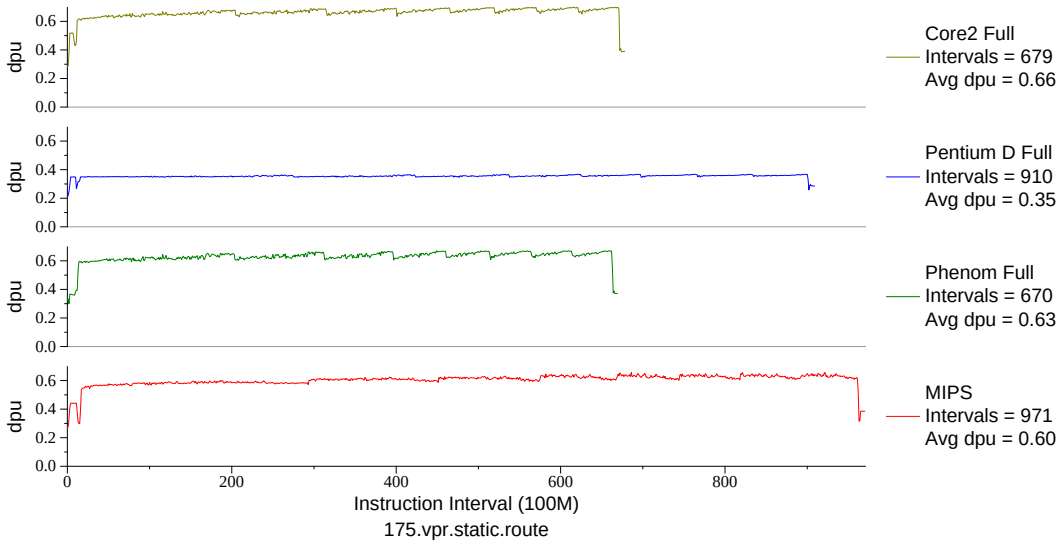

Figure H.11: L1 D$ accesses per μop for `vpr.route` (INT, C, FPGA Place/Route)

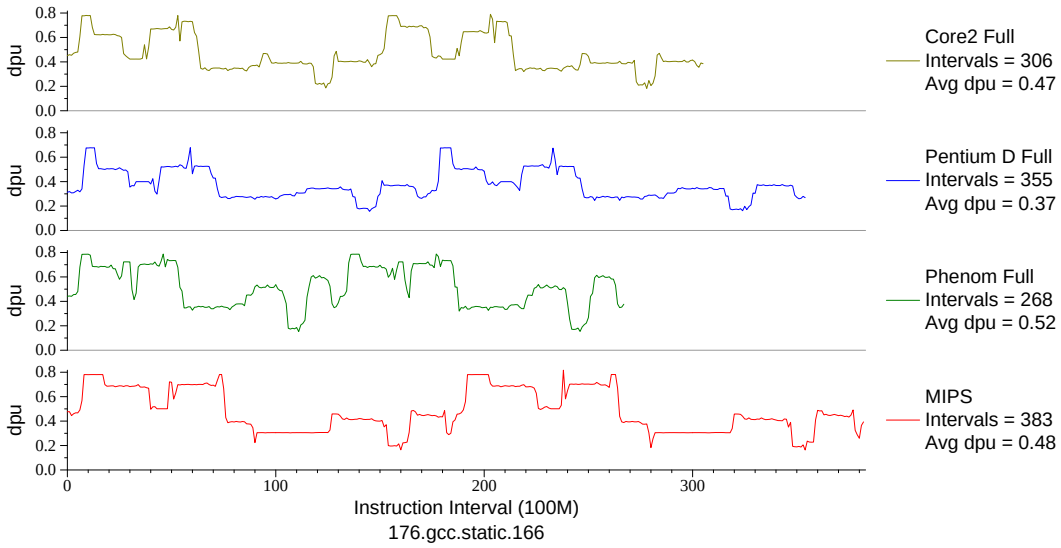

Figure H.12: L1 D$ accesses per μop for `gcc.166` (INT, C, C Compiler)

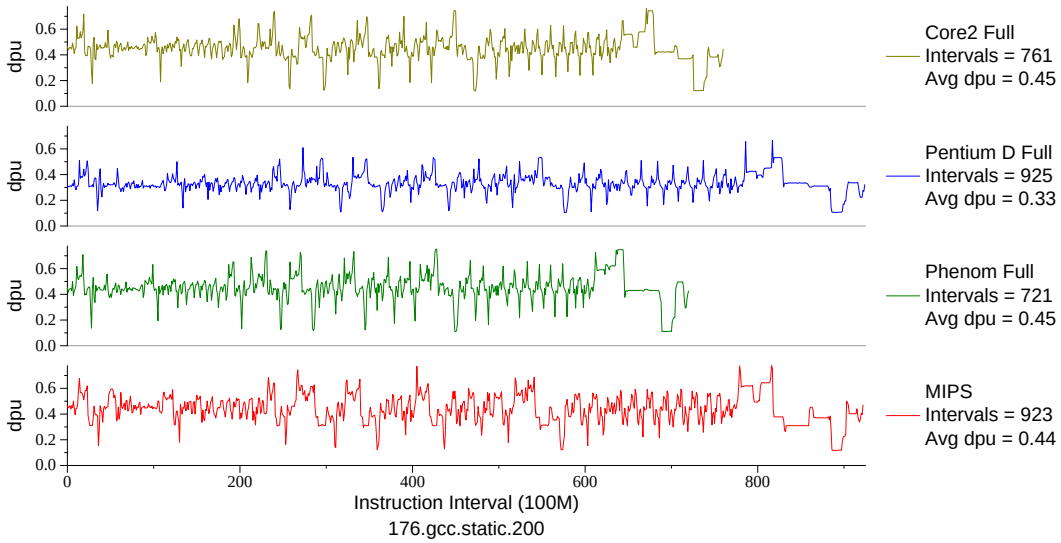

Figure H.13: L1 D$ accesses per μop for `gcc.200` (INT, C, C Compiler)

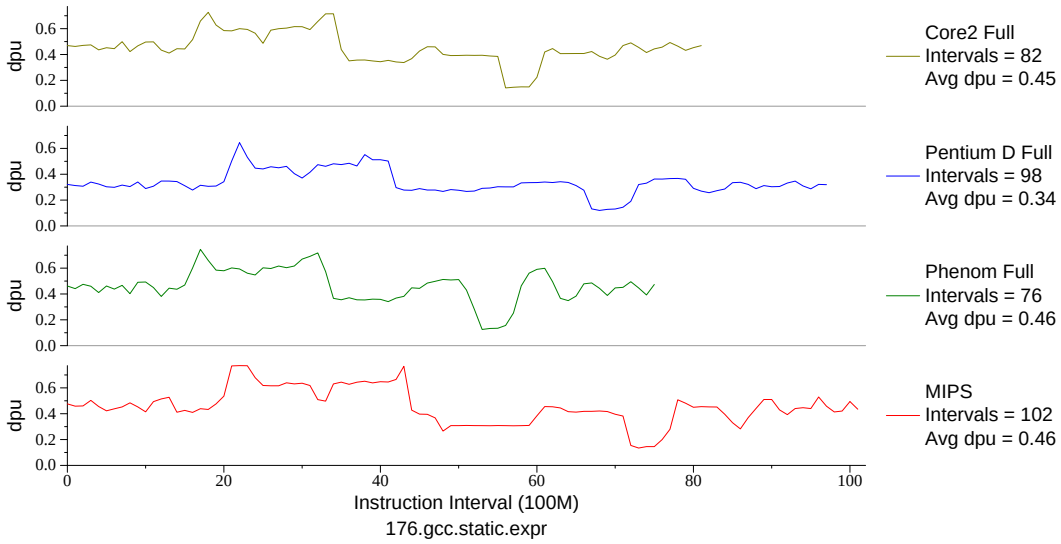

Figure H.14: L1 D$ accesses per μop for `gcc.expr` (INT, C, C Compiler)

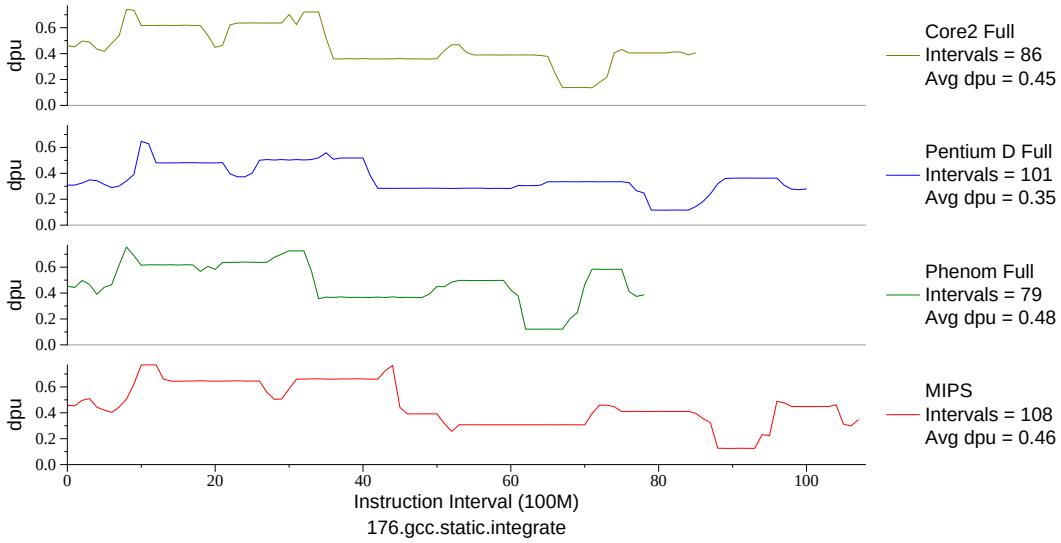

Figure H.15: L1 D$ accesses per μop for `gcc.int` (INT, C, C Compiler)

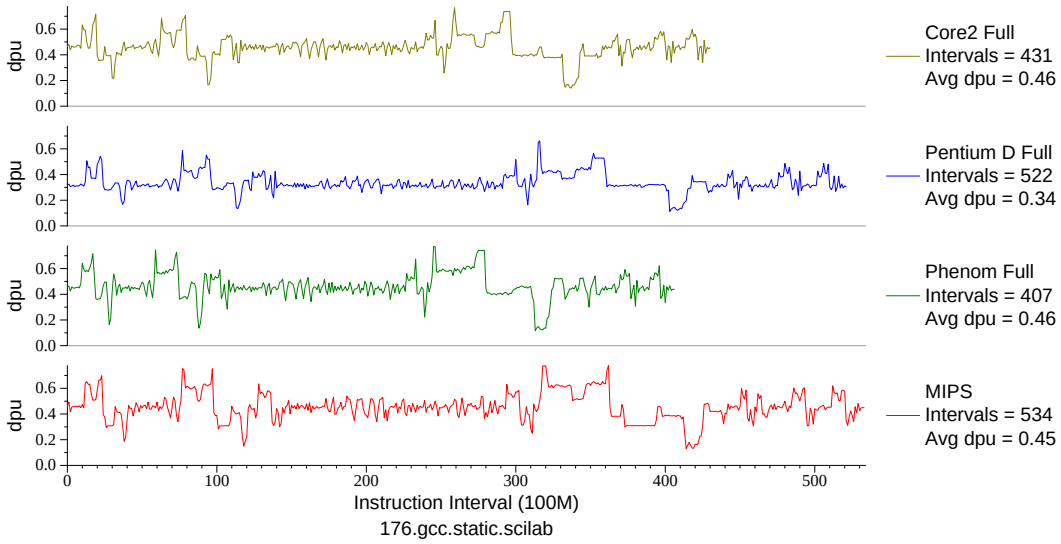

Figure H.16: L1 D$ accesses per μop for `gcc.sci` (INT, C, C Compiler)

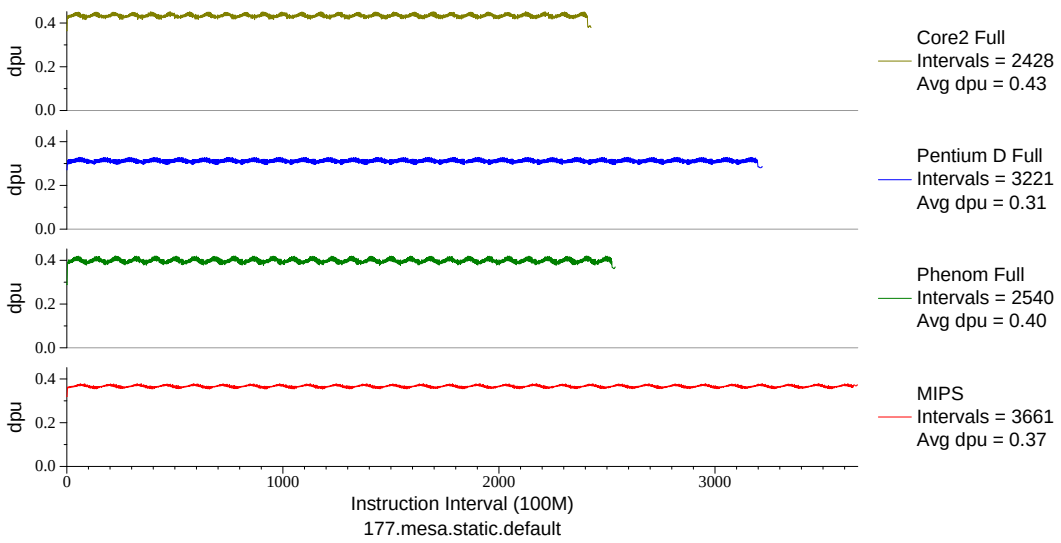

Figure H.17: L1 D$ accesses per μop for `mesa` (FP, C, 3D-graphics)

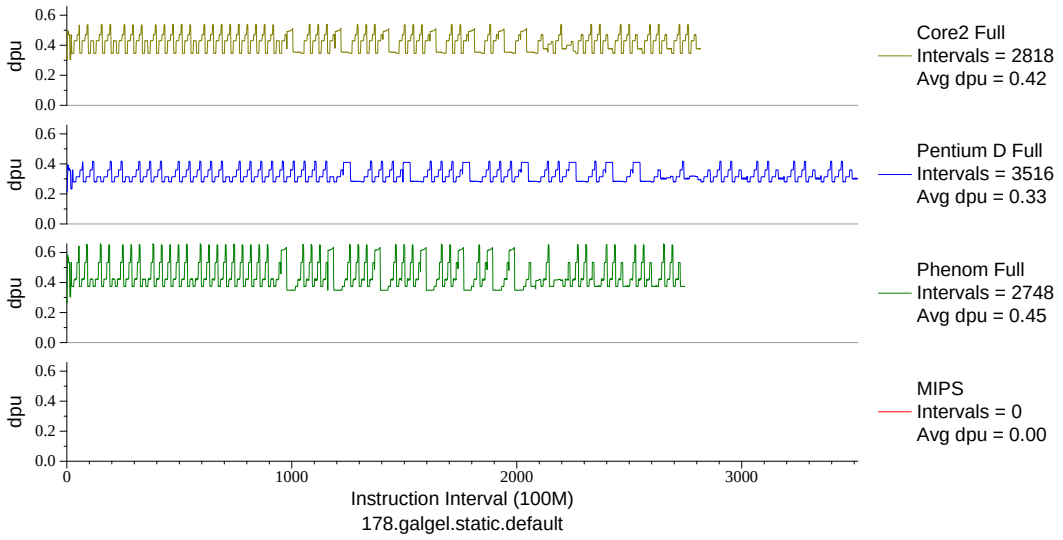

Figure H.18: L1 D$ accesses per μop for `galgel` (FP, F90, Fluid Dynamics)

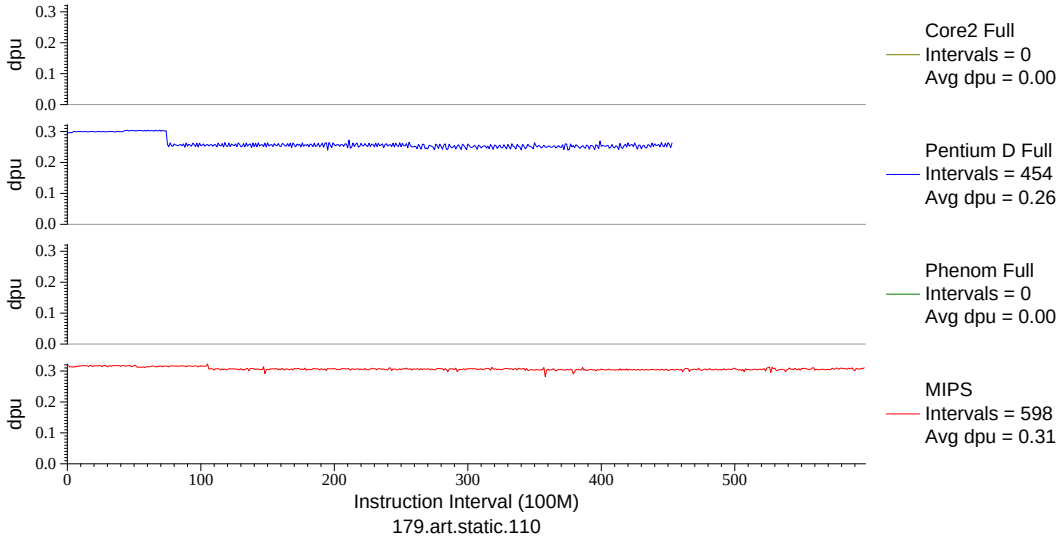

Figure H.19: L1 D$ accesses per μop for `art.110` (FP, C, Neural Networks)

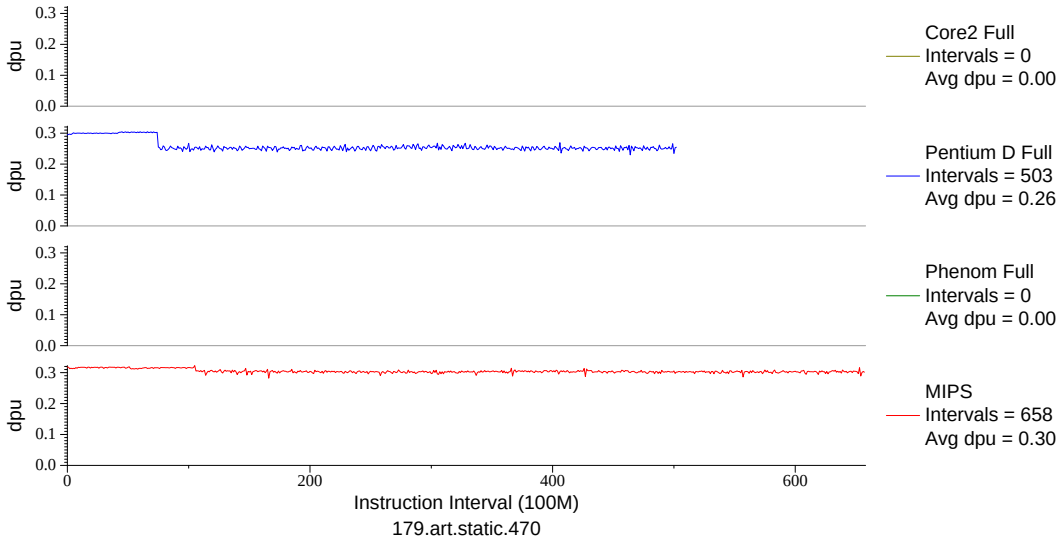

Figure H.20: L1 D$ accesses per μop for `art.470` (FP, C, Neural Networks)

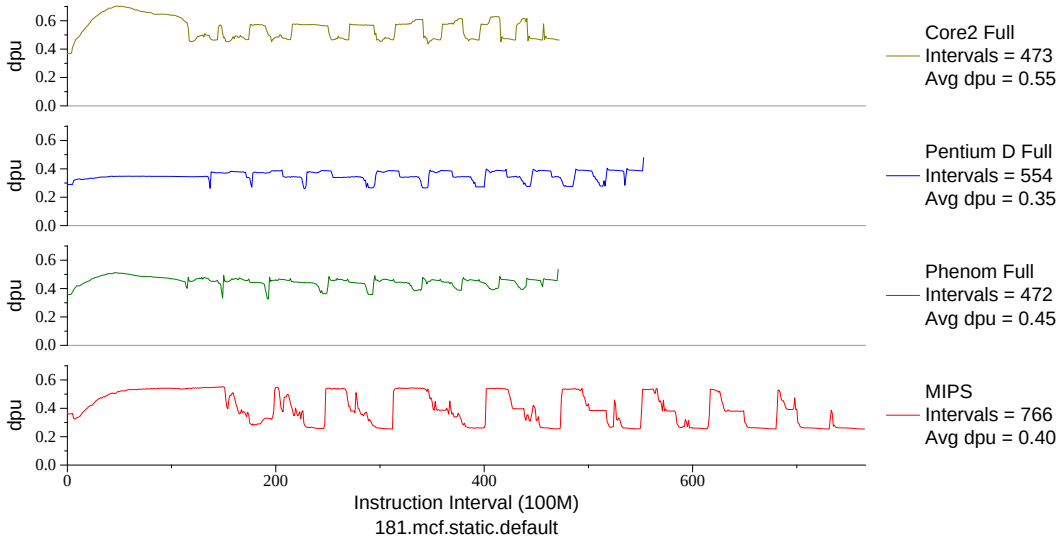

Figure H.21: L1 D$ accesses per μop for `mcf` (INT, C, Combinatorial Opt)

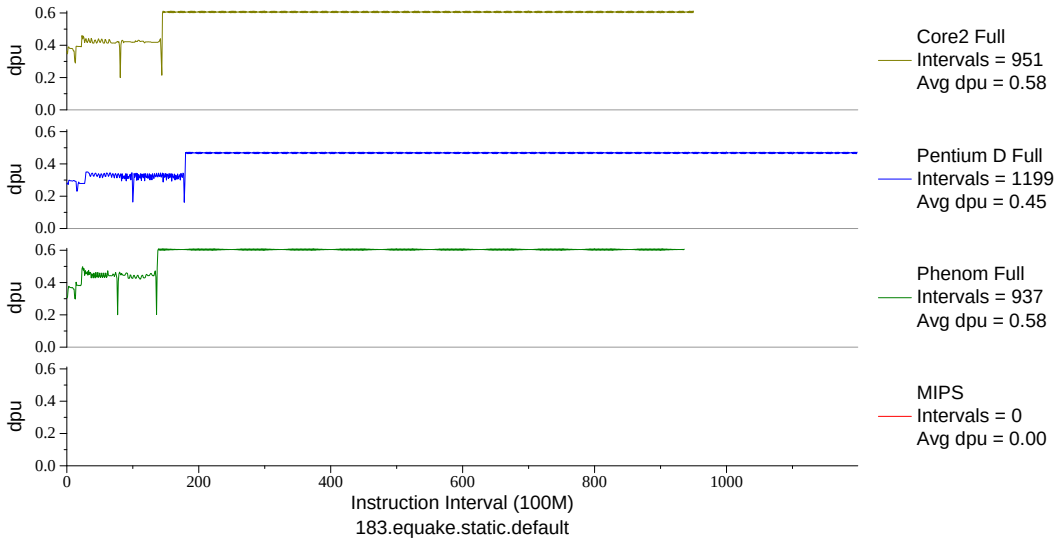

Figure H.22: L1 D$ accesses per μop for `equake` (FP, C, Seismic Propogation)

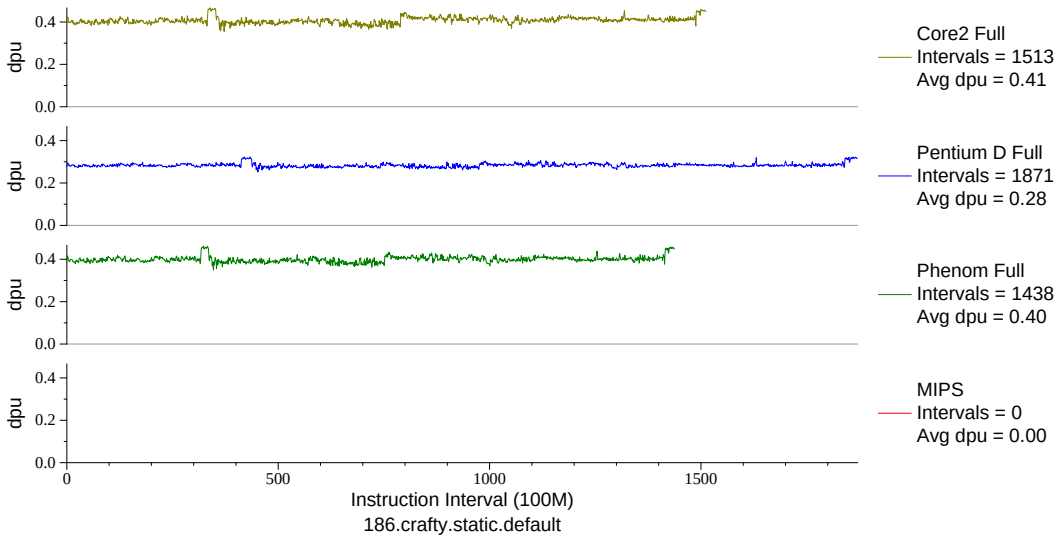

Figure H.23: L1 D$ accesses per μop for `crafty` (INT, C, Chess)

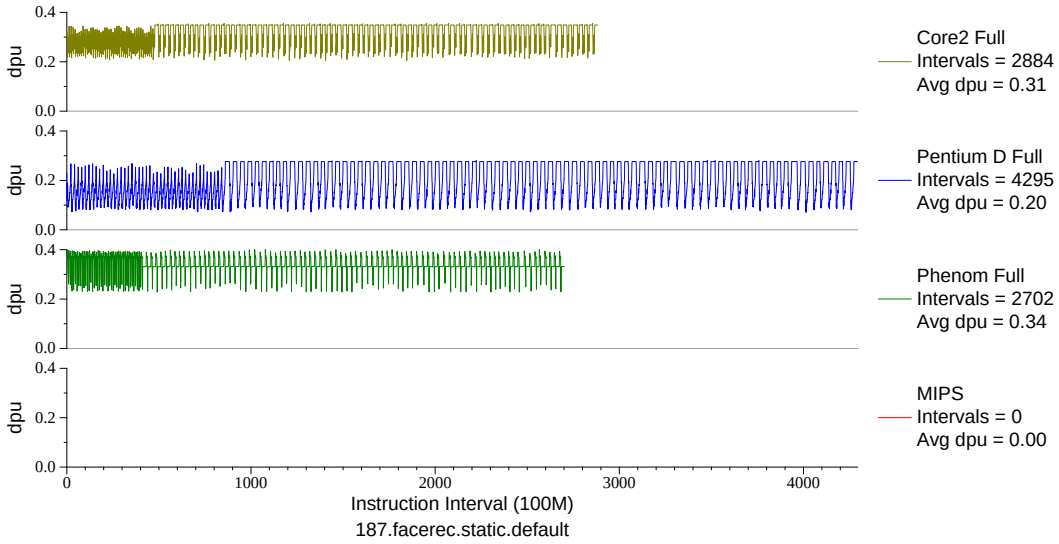

Figure H.24: L1 D$ accesses per μop for `facerec` (FP, F90, Facial Recognition)

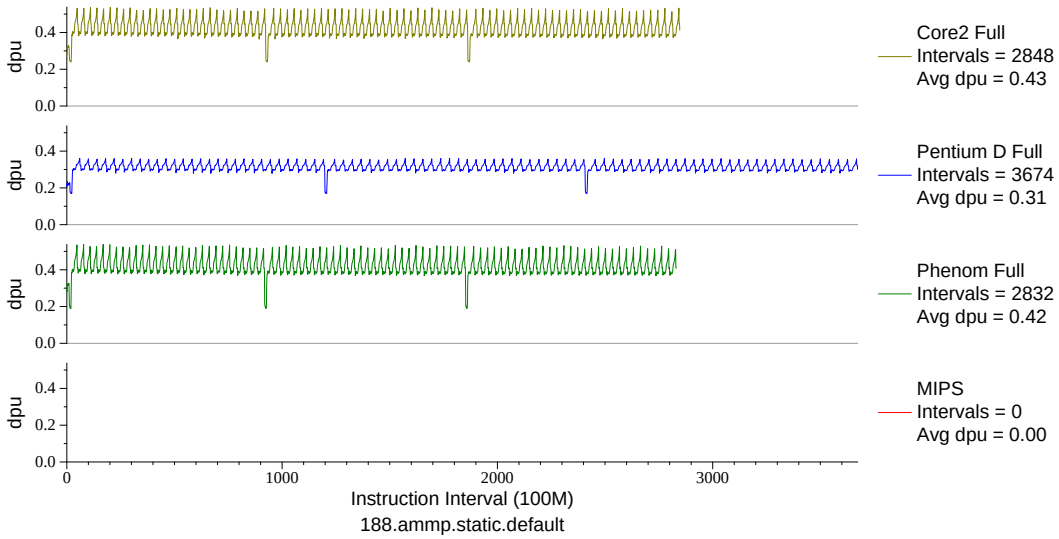

Figure H.25: L1 D$ accesses per μop for `ammp` (FP, C, Chemistry)

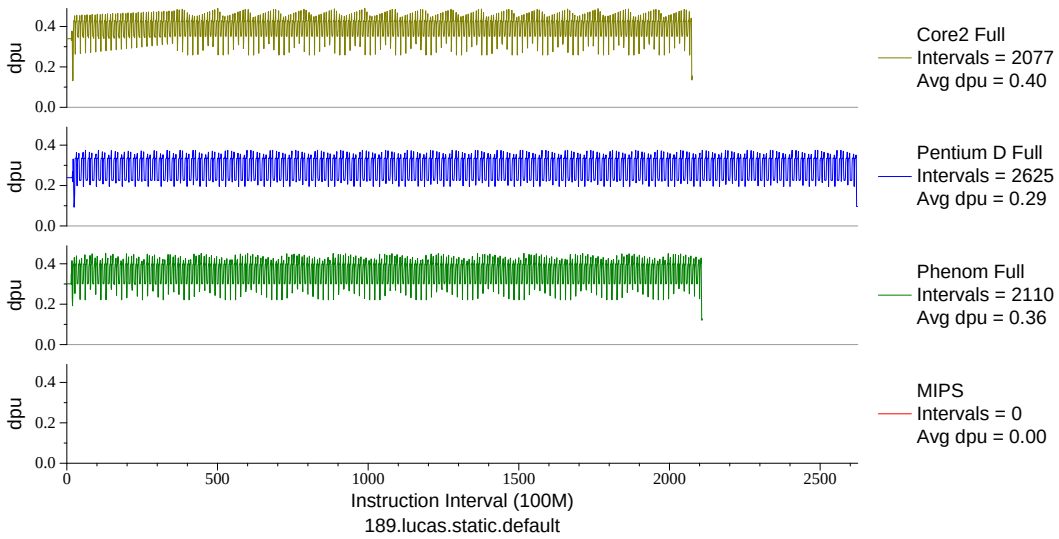

Figure H.26: L1 D$ accesses per μop for `lucas` (FP, F90, Number Theory)

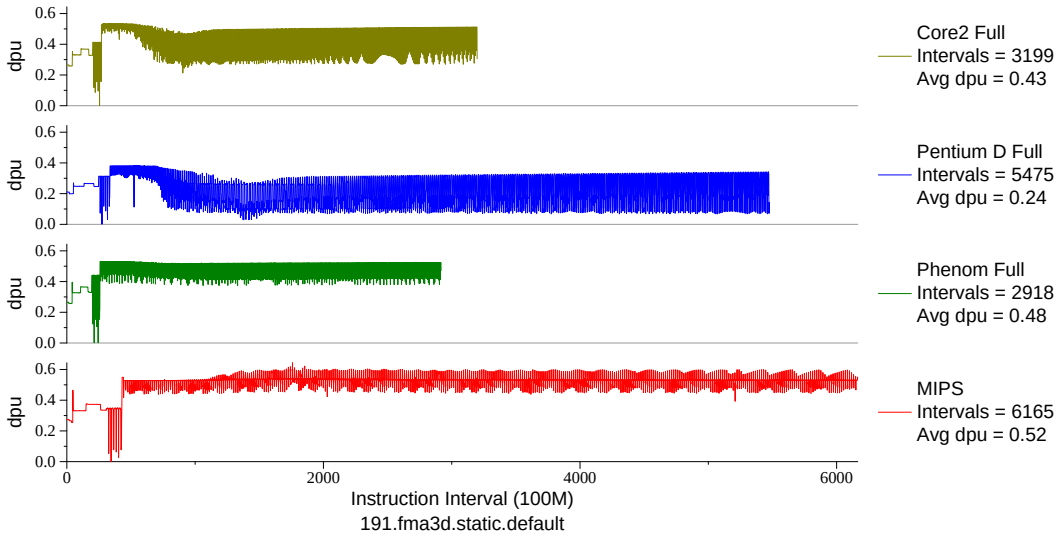

Figure H.27: L1 D$ accesses per μop for `fma3d` (FP, F90, Crash Simulation)

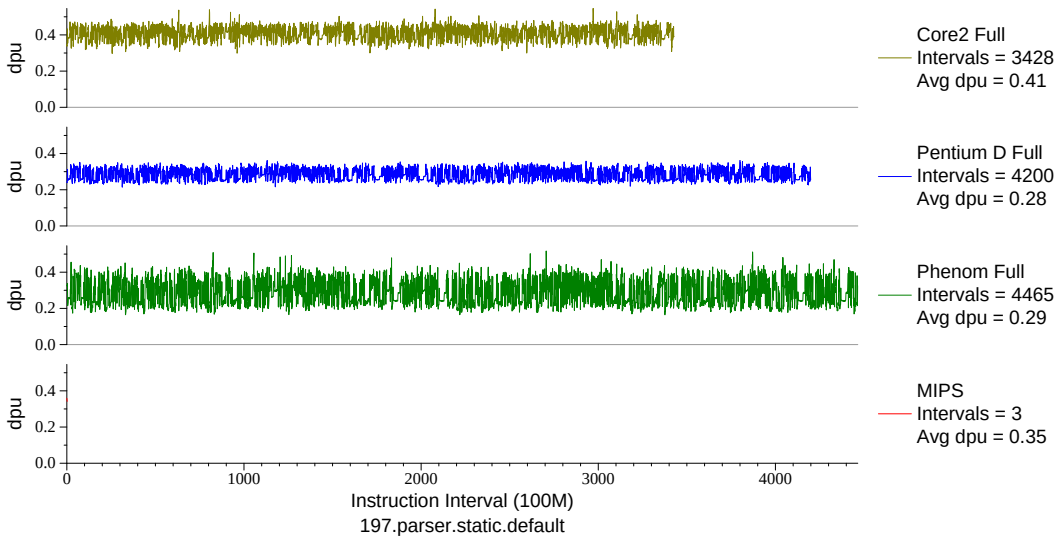

Figure H.28: L1 D$ accesses per μop for `parser` (INT, C, Word Processing)

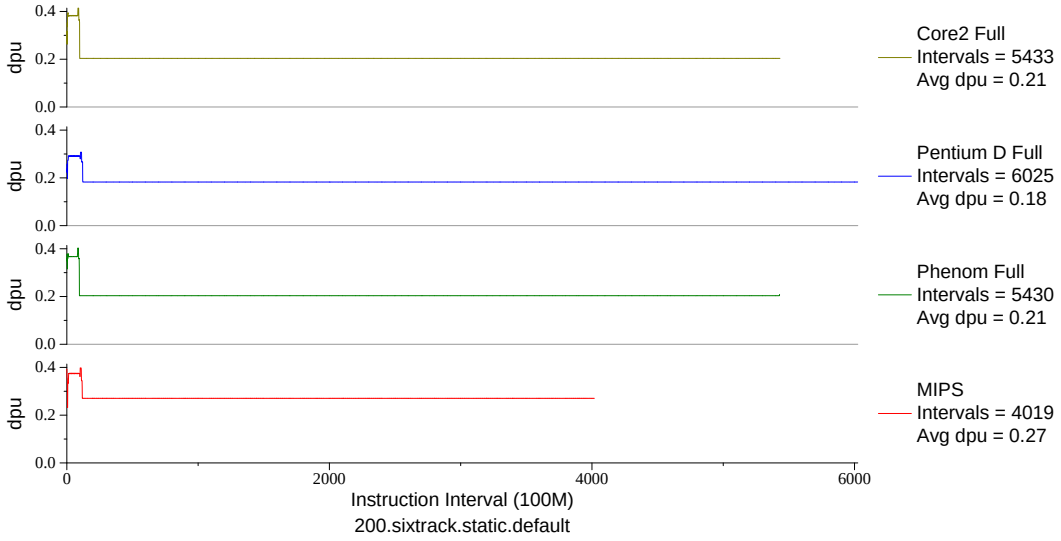

Figure H.29: L1 D$ accesses per μop for `sixtrack` (FP, F77, Nuclear Physics)

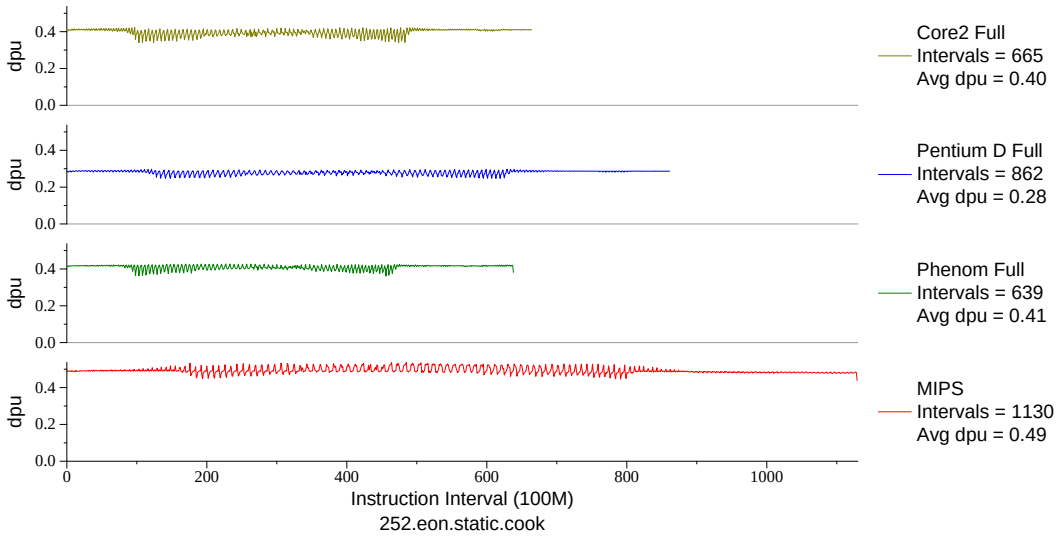

Figure H.30: L1 D$ accesses per μop for `eon.cook` (INT, C++, Computer Graphics)

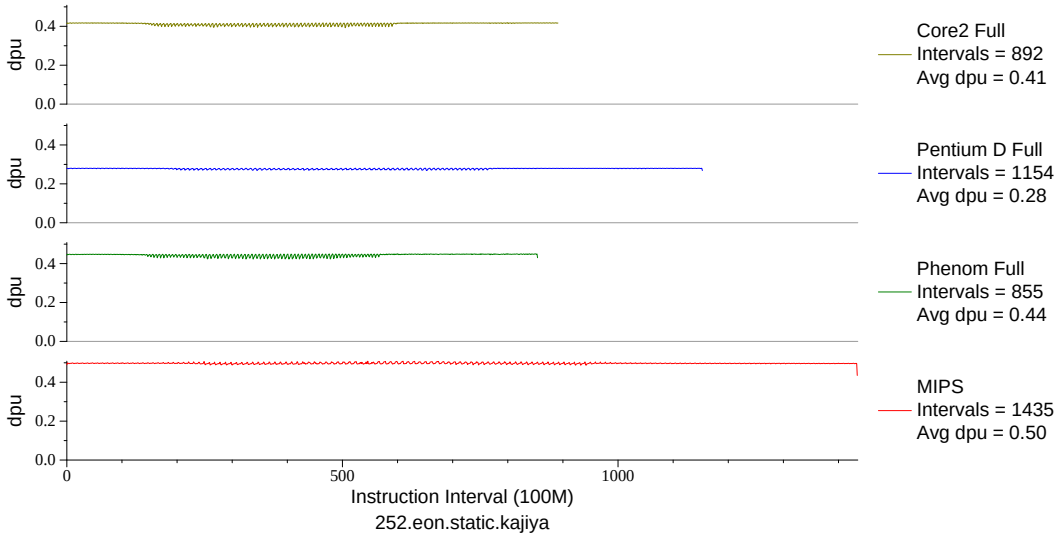

Figure H.31: L1 D$ accesses per μop for `eon.kaj` (INT, C++, Computer Graphics)

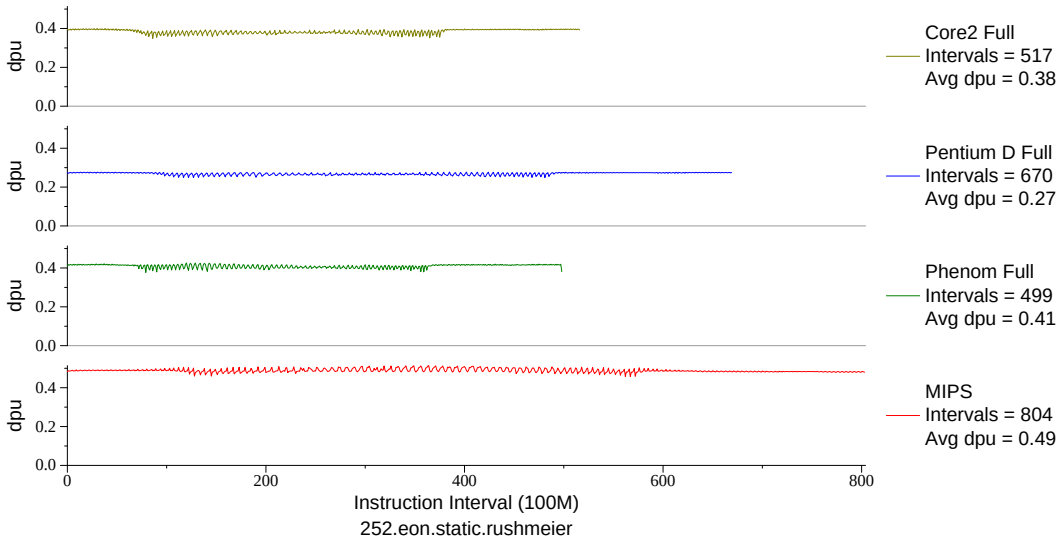

Figure H.32: L1 D$ accesses per μop for `eon.rush` (INT, C++, Computer Graphics)

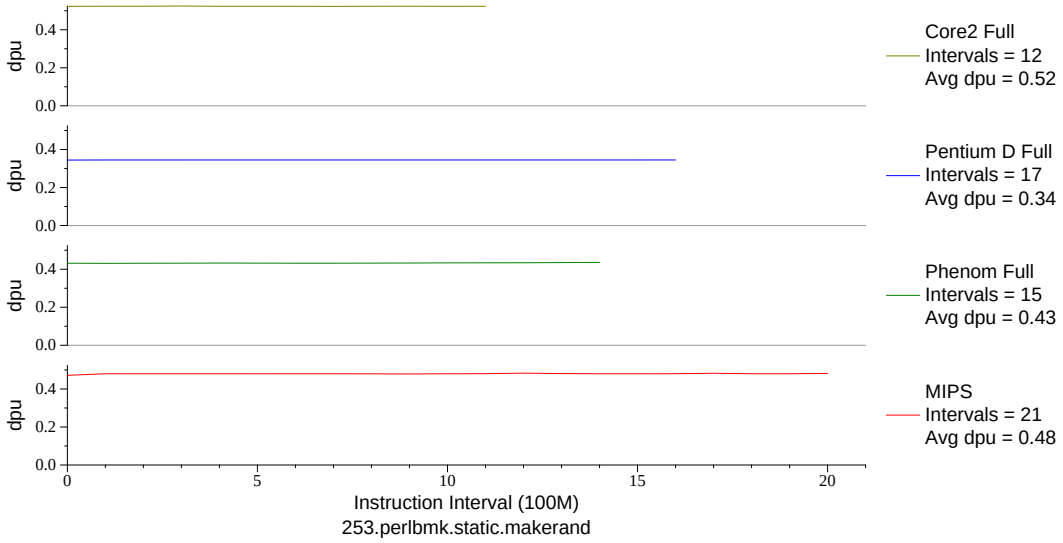

Figure H.33: L1 D$ accesses per μop for `perlbmk.mkrnd` (INT, C, Scripting Language)

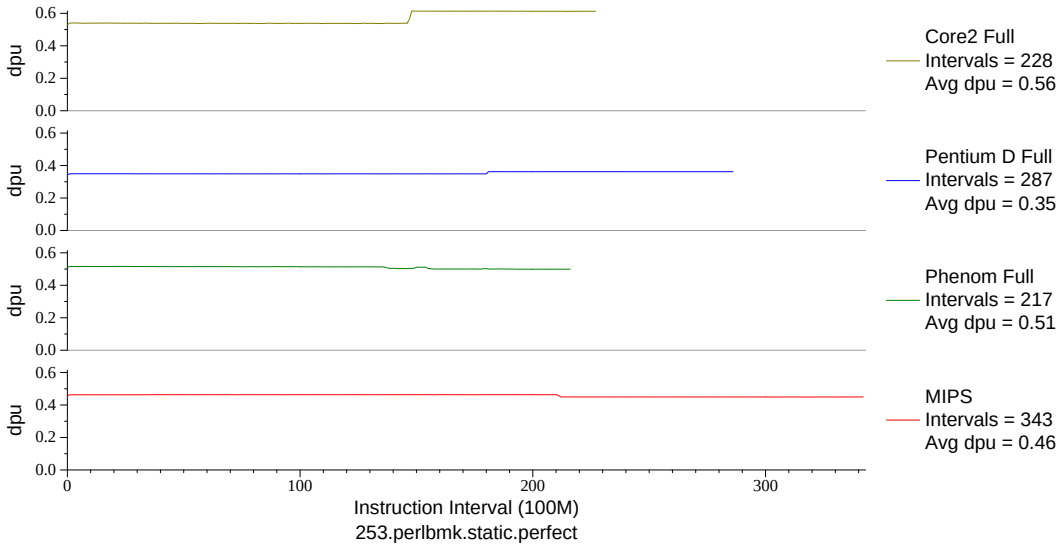

Figure H.34: L1 D$ accesses per μop for `perlbmk.perf` (INT, C, Scripting Language)

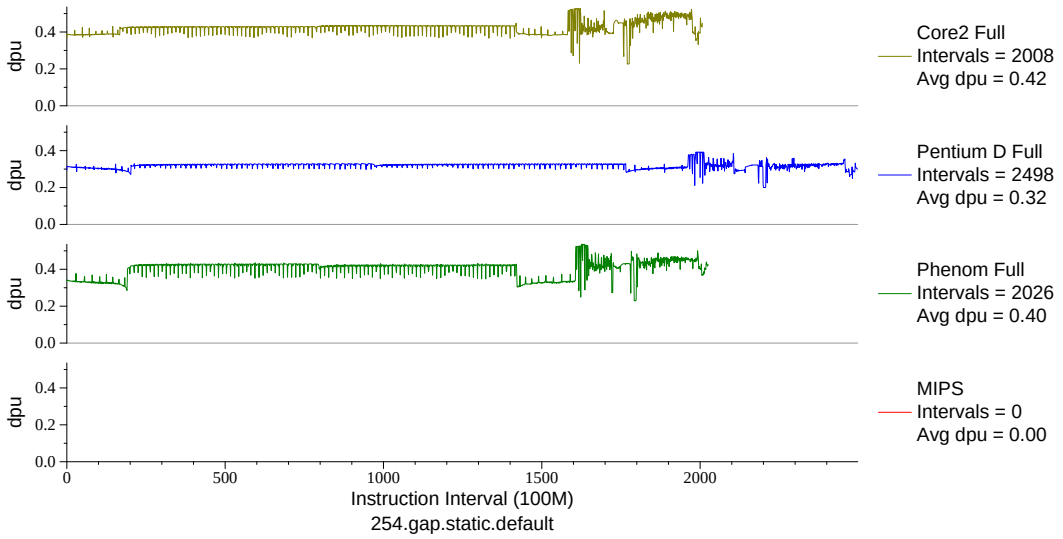

Figure H.35: L1 D$ accesses per μop for `gap` (INT, C, Group Theory)

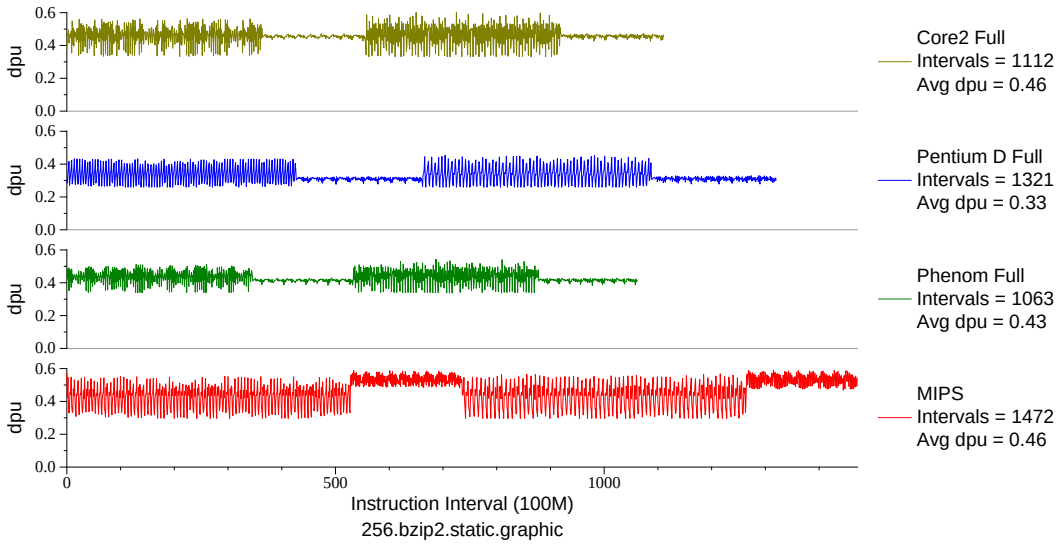

Figure H.36: L1 D$ accesses per μop for `bzip2.graph` (INT, C, Compression)

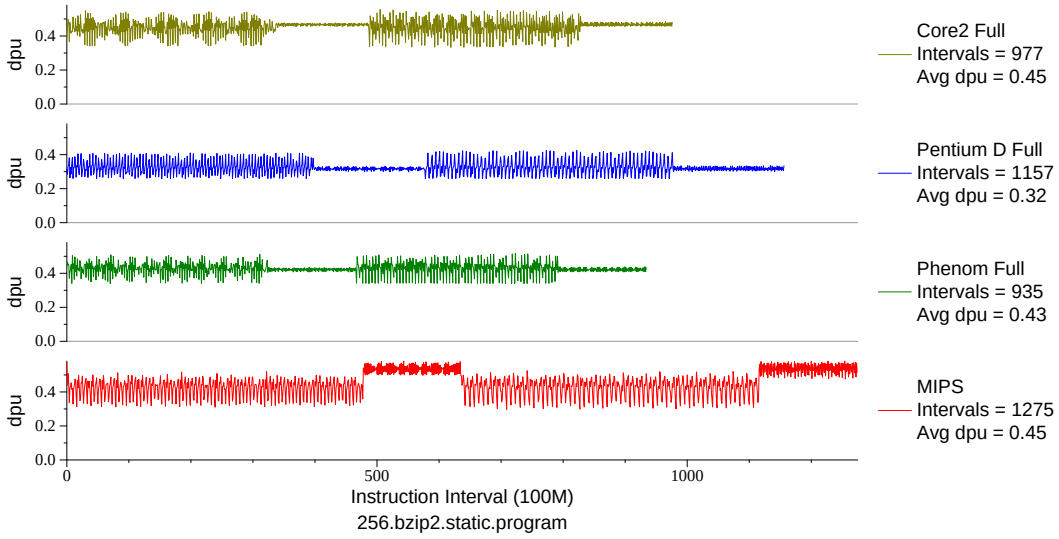

Figure H.37: L1 D$ accesses per μop for `bzip2.prog` (INT, C, Compression)

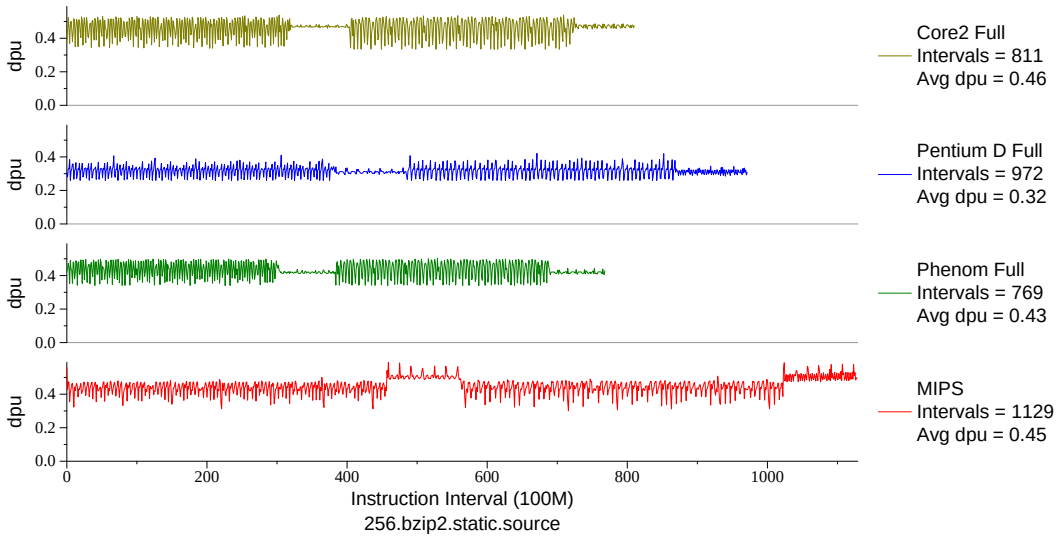

Figure H.38: L1 D$ accesses per μop for `bzip2.src` (INT, C, Compression)

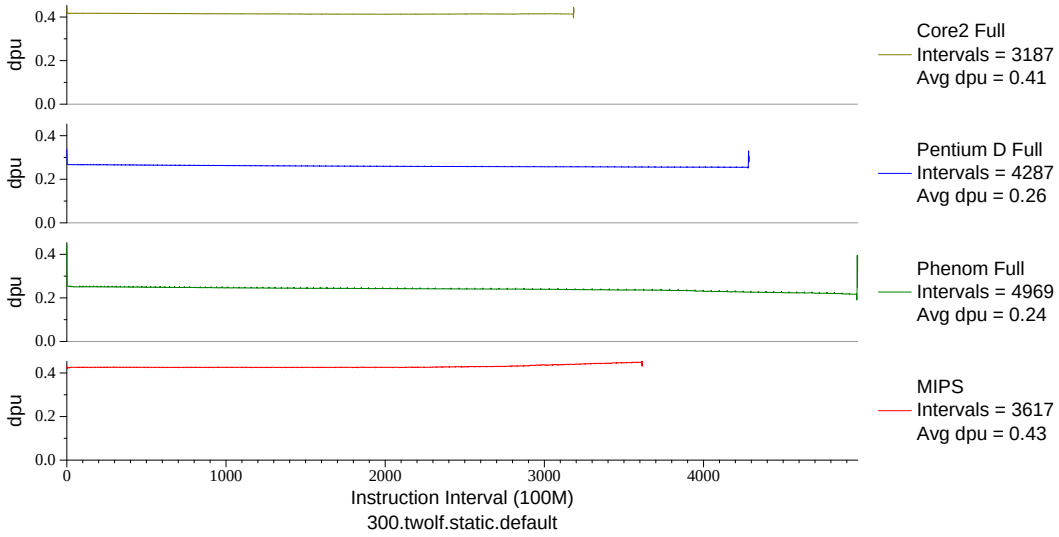

Figure H.39: L1 D$ accesses per μop for `twolf` (INT, C, Place/Route)

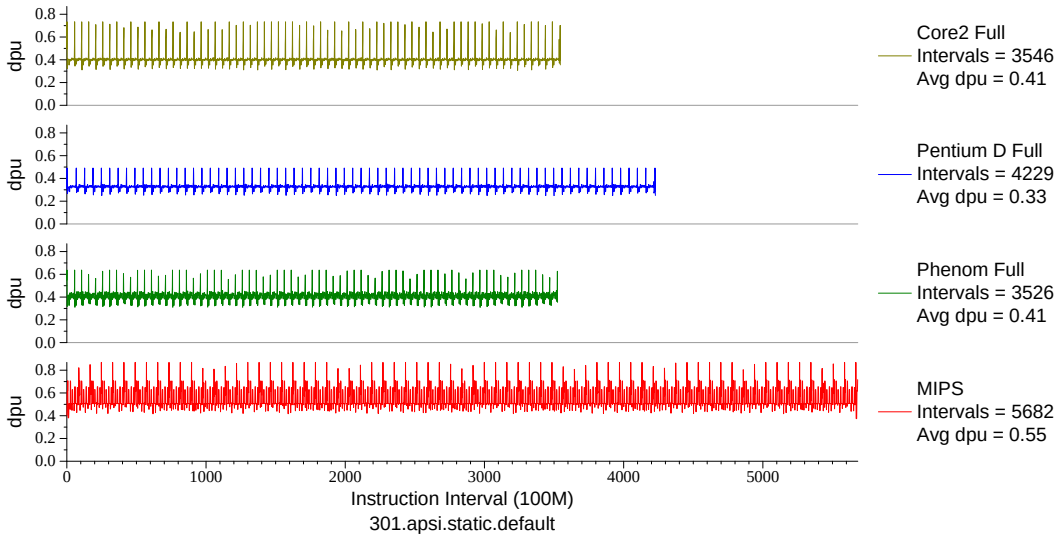

Figure H.40: L1 D$ accesses per μop for `apsi` (FP, F77, Meteorology/Pollution)

APPENDIX I

VALGRIND EXP-BBV TOOL CODE LISTING

Here is the BBV generating plugin for Valgrind, as found in Valgrind 3.5.0.

```
//--------------------------------------------------------------------*/
//--- BBV: a SimPoint basic block vector generator      bbv_main.c ---*/
//--------------------------------------------------------------------*/

/*
   This file is part of BBV, a Valgrind tool for generating SimPoint
   basic block vectors.

   Copyright (C) 2006-2009 Vince Weaver
      vince _at_ csl.cornell.edu

   pcfile code is Copyright (C) 2006-2009 Oriol Prat
      oriol.prat _at_ bsc.es

   This program is free software; you can redistribute it and/or
   modify it under the terms of the GNU General Public License as
   published by the Free Software Foundation; either version 2 of the
   License, or (at your option) any later version.

   This program is distributed in the hope that it will be useful, but
   WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
   02111-1307, USA.

   The GNU General Public License is contained in the file COPYING.
*/


#include "pub_tool_basics.h"
#include "pub_tool_tooliface.h"
#include "pub_tool_options.h"    /* command line options */

#include "pub_tool_vki.h"        /* vki_stat */
#include "pub_tool_libcbase.h"   /* VG_(strlen) */
#include "pub_tool_libcfile.h"   /* VG_(write) */
#include "pub_tool_libcprint.h"  /* VG_(printf) */
#include "pub_tool_libcassert.h" /* VG_(exit) */
#include "pub_tool_mallocfree.h" /* plain_free */
#include "pub_tool_machine.h"    /* VG_(fnptr_to_fnentry) */
#include "pub_tool_debuginfo.h"  /* VG_(get_fnname) */

#include "pub_tool_oset.h"       /* ordered set stuff */
```

```
   /* instruction special cases */
#define REP_INSTRUCTION    0x1
#define FLDCW_INSTRUCTION 0x2

   /* interval variables */
#define DEFAULT_GRAIN_SIZE 100000000  /* 100 million by default */
static Int interval_size=DEFAULT_GRAIN_SIZE;

   /* filenames */
static UChar *clo_bb_out_file="bb.out.%p";
static UChar *clo_pc_out_file="pc.out.%p";
static UChar *pc_out_file=NULL;
static UChar *bb_out_file=NULL;


   /* output parameters */
static Bool instr_count_only=False;
static Bool generate_pc_file=False;

   /* write buffer */
static UChar buf[1024];

   /* Global values */
static OSet* instr_info_table;  /* table that holds the basic block info */
static Int block_num=1;         /* global next block number */
static Int current_thread=0;
static Int allocated_threads=1;
struct thread_info *bbv_thread=NULL;

   /* Per-thread variables */
struct thread_info {
   ULong dyn_instr;         /* Current retired instruction count */
   ULong total_instr;       /* Total retired instruction count   */
   Addr last_rep_addr;      /* rep counting values */
   ULong rep_count;
   ULong global_rep_count;
   ULong unique_rep_count;
   ULong fldcw_count;       /* fldcw count */
   Int bbtrace_fd;          /* file descriptor */
};

#define FUNCTION_NAME_LENGTH 20

struct BB_info {
   Addr        BB_addr;           /* used as key, must be first           */
   Int         n_instrs;          /* instructions in the basic block      */
   Int         block_num;         /* unique block identifier              */
   Int         *inst_counter;     /* times entered * num_instructions     */
   Bool        is_entry;          /* is this block a function entry point */
   UChar       fn_name[FUNCTION_NAME_LENGTH];  /* Function block is in    */
};


   /* dump the optional PC file, which contains basic block number to */
   /*    instruction address and function name mappings               */
```

```
static void dumpPcFile(void)
{
   struct BB_info    *bb_elem;
   Int               pctrace_fd;
   SysRes            sres;

   pc_out_file =
          VG_(expand_file_name)("--pc-out-file", clo_pc_out_file);

   sres = VG_(open)(pc_out_file, VKI_O_CREAT|VKI_O_TRUNC|VKI_O_WRONLY,
                                 VKI_S_IRUSR|VKI_S_IWUSR|VKI_S_IRGRP|VKI_S_IWGRP);
   if (sr_isError(sres)) {
      VG_(umsg)("Error: cannot create pc file %s\n", pc_out_file);
      VG_(exit)(1);
   } else {
      pctrace_fd = sr_Res(sres);
   }

      /* Loop through the table, printing the number, address, */
      /*    and function name for each basic block             */
   VG_(OSetGen_ResetIter)(instr_info_table);
   while ( (bb_elem = VG_(OSetGen_Next)(instr_info_table)) ) {
      VG_(write)(pctrace_fd,"F",1);
      VG_(sprintf)( buf,":%d:%x:%s\n",
                        bb_elem->block_num,
                        (Int)bb_elem->BB_addr,
                        bb_elem->fn_name);
      VG_(write)(pctrace_fd, (void*)buf, VG_(strlen)(buf));
   }

   VG_(close)(pctrace_fd);
}

static Int open_tracefile(Int thread_num)
{
   SysRes  sres;
   UChar temp_string[2048];

      /* For thread 1, don't append any thread number  */
      /* This lets the single-thread case not have any */
      /* extra values appended to the file name.       */
   if (thread_num==1) {
      VG_(strncpy)(temp_string,bb_out_file,2047);
   }
   else {
      VG_(sprintf)(temp_string,"%s.%d",bb_out_file,thread_num);
   }

   sres = VG_(open)(temp_string, VKI_O_CREAT|VKI_O_TRUNC|VKI_O_WRONLY,
                                 VKI_S_IRUSR|VKI_S_IWUSR|VKI_S_IRGRP|VKI_S_IWGRP);

   if (sr_isError(sres)) {
      VG_(umsg)("Error: cannot create bb file %s\n",temp_string);
      VG_(exit)(1);
   }
```

```
   return sr_Res(sres);
}

static void handle_overflow(void)
{
   struct BB_info *bb_elem;

   if (bbv_thread[current_thread].dyn_instr > interval_size) {

      if (!instr_count_only) {

         /* If our output fd hasn't been opened, open it */
         if (bbv_thread[current_thread].bbtrace_fd < 0) {
            bbv_thread[current_thread].bbtrace_fd=open_tracefile(current_thread);
         }

        /* put an entry to the bb.out file */

         VG_(write)(bbv_thread[current_thread].bbtrace_fd,"T",1);

         VG_(OSetGen_ResetIter)(instr_info_table);
         while ( (bb_elem = VG_(OSetGen_Next)(instr_info_table)) ) {
            if ( bb_elem->inst_counter[current_thread] != 0 ) {
               VG_(sprintf)( buf,":%d:%d   ",
                           bb_elem->block_num,
                           bb_elem->inst_counter[current_thread]);
               VG_(write)(bbv_thread[current_thread].bbtrace_fd,
                           (void*)buf, VG_(strlen)(buf));
               bb_elem->inst_counter[current_thread] = 0;
            }
         }

         VG_(write)(bbv_thread[current_thread].bbtrace_fd,"\n",1);
      }

      bbv_thread[current_thread].dyn_instr -= interval_size;
   }
}


static void close_out_reps(void)
{
   bbv_thread[current_thread].global_rep_count+=bbv_thread[current_thread].rep_count;
   bbv_thread[current_thread].unique_rep_count++;
   bbv_thread[current_thread].rep_count=0;
}

   /* Generic function to get called each instruction */
static VG_REGPARM(1) void per_instruction_BBV(struct BB_info *bbInfo)
{
   Int n_instrs=1;

   tl_assert(bbInfo);
```

```
      /* we finished rep but didn't clear out count */
   if (bbv_thread[current_thread].rep_count) {
      n_instrs++;
      close_out_reps();
   }

   bbInfo->inst_counter[current_thread]+=n_instrs;

   bbv_thread[current_thread].total_instr+=n_instrs;
   bbv_thread[current_thread].dyn_instr +=n_instrs;

   handle_overflow();
}

   /* Function to get called if instruction has a rep prefix */
static VG_REGPARM(1) void per_instruction_BBV_rep(Addr addr)
{
      /* handle back-to-back rep instructions */
   if (bbv_thread[current_thread].last_rep_addr!=addr) {
      if (bbv_thread[current_thread].rep_count) {
         close_out_reps();
         bbv_thread[current_thread].total_instr++;
         bbv_thread[current_thread].dyn_instr++;
      }
      bbv_thread[current_thread].last_rep_addr=addr;
   }

   bbv_thread[current_thread].rep_count++;

}

   /* Function to call if our instruction has a fldcw instruction */
static VG_REGPARM(1) void per_instruction_BBV_fldcw(struct BB_info *bbInfo)
{
   Int n_instrs=1;

   tl_assert(bbInfo);

      /* we finished rep but didn't clear out count */
   if (bbv_thread[current_thread].rep_count) {
      n_instrs++;
      close_out_reps();
   }

      /* count fldcw instructions */
   bbv_thread[current_thread].fldcw_count++;

   bbInfo->inst_counter[current_thread]+=n_instrs;

   bbv_thread[current_thread].total_instr+=n_instrs;
   bbv_thread[current_thread].dyn_instr +=n_instrs;

   handle_overflow();
}
```

```
   /* Check if the instruction pointed to is one that needs */
   /*    special handling.  If so, set a bit in the return    */
   /*    value indicating what type.                          */
static Int get_inst_type(Int len, Addr addr)
{
   int result=0;

#if defined(VGA_x86) || defined(VGA_amd64)

   unsigned char *inst_pointer;
   unsigned char inst_byte;
   int i,possible_rep;

   /* rep prefixed instructions are counted as one instruction on */
   /*      x86 processors and must be handled as a special case    */

   /* Also, the rep prefix is re-used as part of the opcode for   */
   /*      SSE instructions.  So we need to specifically check for */
   /*      the following: movs, cmps, scas, lods, stos, ins, outs  */

   inst_pointer=(unsigned char *)addr;
   i=0;
   inst_byte=0;
   possible_rep=0;

   while (i<len) {

      inst_byte=*inst_pointer;

      if ( (inst_byte == 0x67) ||            /* size override prefix */
           (inst_byte == 0x66) ||            /* size override prefix */
           (inst_byte == 0x48) ) {           /* 64-bit prefix */
      } else if ( (inst_byte == 0xf2) ||     /* rep prefix    */
                  (inst_byte == 0xf3) ) {    /* repne prefix  */
         possible_rep=1;
      } else {
         break;                              /* other byte, exit */
      }

      i++;
      inst_pointer++;
   }

   if ( possible_rep &&
        ( ( (inst_byte >= 0xa4) &&     /* movs,cmps,scas */
            (inst_byte <= 0xaf) ) ||   /* lods,stos      */
          ( (inst_byte >= 0x6c) &&
            (inst_byte <= 0x6f) ) ) ) {  /* ins,outs       */

      result|=REP_INSTRUCTION;
   }

   /* fldcw instructions are double-counted by the hardware       */
   /*      performance counters on pentium 4 processors so it is   */
   /*      useful to have that count when doing validation work.   */
```

```
   inst_pointer=(unsigned char *)addr;
   if (len>1) {
         /* FLDCW detection */
         /* opcode is 0xd9/5, ie 1101 1001 oo10 1mmm */
      if ((*inst_pointer==0xd9) &&
          (*(inst_pointer+1)<0xb0) &&  /* need this case of fldz, etc, count */
          ( (*(inst_pointer+1) & 0x38) == 0x28)) {
         result|=FLDCW_INSTRUCTION;
      }
   }

#endif
   return result;
}



   /* Our instrumentation function        */
   /*    sbIn = super block to translate  */
   /*    layout = guest layout            */
   /*    gWordTy = size of guest word     */
   /*    hWordTy = size of host word      */
static IRSB* bbv_instrument ( VgCallbackClosure* closure,
                              IRSB* sbIn, VexGuestLayout* layout,
                              VexGuestExtents* vge,
                              IRType gWordTy, IRType hWordTy )
{
   Int       i,n_instrs=1;
   IRSB      *sbOut;
   IRStmt    *st;
   struct BB_info  *bbInfo;
   Addr64    origAddr,ourAddr;
   IRDirty   *di;
   IRExpr    **argv, *arg1;
   Int       regparms,opcode_type;

      /* We don't handle a host/guest word size mismatch */
   if (gWordTy != hWordTy) {
      VG_(tool_panic)("host/guest word size mismatch");
   }

      /* Set up SB */
   sbOut = deepCopyIRSBExceptStmts(sbIn);

      /* Copy verbatim any IR preamble preceding the first IMark */
   i = 0;
   while ( (i < sbIn->stmts_used) && (sbIn->stmts[i]->tag!=Ist_IMark)) {
      addStmtToIRSB( sbOut, sbIn->stmts[i] );
      i++;
   }

      /* Get the first statement */
   tl_assert(sbIn->stmts_used > 0);
   st = sbIn->stmts[i];
```

```
   /* double check we are at a Mark statement */
tl_assert(Ist_IMark == st->tag);

origAddr=st->Ist.IMark.addr;

   /* Get the BB_info */
bbInfo = VG_(OSetGen_Lookup)(instr_info_table, &origAddr);

if (bbInfo==NULL) {

      /* BB never translated before (at this address, at least;          */
      /* could have been unloaded and then reloaded elsewhere in memory) */

      /* allocate and initialize a new basic block structure */
   bbInfo=VG_(OSetGen_AllocNode)(instr_info_table, sizeof(struct BB_info));
   bbInfo->BB_addr = origAddr;
   bbInfo->n_instrs = n_instrs;
   bbInfo->inst_counter=VG_(calloc)("bbv_instrument",
                                    allocated_threads,
                                    sizeof(Int));

      /* assign a unique block number */
   bbInfo->block_num=block_num;
   block_num++;
      /* get function name and entry point information */
   VG_(get_fnname)(origAddr,bbInfo->fn_name,FUNCTION_NAME_LENGTH);
   bbInfo->is_entry=VG_(get_fnname_if_entry)(origAddr, bbInfo->fn_name,
                                             FUNCTION_NAME_LENGTH);
      /* insert structure into table */
   VG_(OSetGen_Insert)( instr_info_table, bbInfo );
}

   /* Iterate through the basic block, putting the original   */
   /* instructions in place, plus putting a call to updateBBV */
   /* for each original instruction                           */

   /* This is less efficient than only instrumenting the BB   */
   /* But it gives proper results given the fact that         */
   /* valgrind uses superblocks (not basic blocks) by default */


while(i < sbIn->stmts_used) {
   st=sbIn->stmts[i];

   if (st->tag == Ist_IMark) {

      ourAddr = st->Ist.IMark.addr;

      opcode_type=get_inst_type(st->Ist.IMark.len,ourAddr);

      regparms=1;
      arg1= mkIRExpr_HWord( (HWord)bbInfo);
      argv= mkIRExprVec_1(arg1);
```

```
         if (opcode_type&REP_INSTRUCTION) {
            arg1= mkIRExpr_HWord(ourAddr);
            argv= mkIRExprVec_1(arg1);
            di= unsafeIRDirty_0_N( regparms, "per_instruction_BBV_rep",
                                   VG_(fnptr_to_fnentry)( &per_instruction_BBV_rep ),
                                   argv);
         }
         else if (opcode_type&FLDCW_INSTRUCTION) {
            di= unsafeIRDirty_0_N( regparms, "per_instruction_BBV_fldcw",
                                   VG_(fnptr_to_fnentry)( &per_instruction_BBV_fldcw ),
                                   argv);
         }
         else {
         di= unsafeIRDirty_0_N( regparms, "per_instruction_BBV",
                                   VG_(fnptr_to_fnentry)( &per_instruction_BBV ),
                                   argv);
         }


            /* Insert our call */
         addStmtToIRSB( sbOut,  IRStmt_Dirty(di));
      }

         /* Insert the original instruction */
      addStmtToIRSB( sbOut, st );

      i++;
   }

   return sbOut;
}

static struct thread_info *allocate_new_thread(struct thread_info *old,
                                    Int old_number, Int new_number)
{
   struct thread_info *temp;
   struct BB_info   *bb_elem;
   Int i;

   temp=VG_(realloc)("bbv_main.c_allocate_threads",
                     old,
                     new_number*sizeof(struct thread_info));

      /* init the new thread */
      /* We loop in case the new thread is not contiguous */
   for(i=old_number;i<new_number;i++) {
      temp[i].last_rep_addr=0;
      temp[i].dyn_instr=0;
      temp[i].total_instr=0;
      temp[i].global_rep_count=0;
      temp[i].unique_rep_count=0;
      temp[i].rep_count=0;
      temp[i].fldcw_count=0;
      temp[i].bbtrace_fd=-1;
```

```
   }
      /* expand the inst_counter on all allocated basic blocks */
   VG_(OSetGen_ResetIter)(instr_info_table);
   while ( (bb_elem = VG_(OSetGen_Next)(instr_info_table)) ) {
      bb_elem->inst_counter =
                  VG_(realloc)("bbv_main.c_inst_counter",
                               bb_elem->inst_counter,
                               new_number*sizeof(Int));
      for(i=old_number;i<new_number;i++) {
         bb_elem->inst_counter[i]=0;
      }
   }

   return temp;
}

static void bbv_thread_called ( ThreadId tid, ULong nDisp )
{
   if (tid >= allocated_threads) {
      bbv_thread=allocate_new_thread(bbv_thread,allocated_threads,tid+1);
      allocated_threads=tid+1;
   }
   current_thread=tid;
}




/*--------------------------------------------------------------------*/
/*--- Setup                                                        ---*/
/*--------------------------------------------------------------------*/

static void bbv_post_clo_init(void)
{
   bb_out_file =
         VG_(expand_file_name)("--bb-out-file", clo_bb_out_file);

      /* Try a closer approximation of basic blocks  */
      /* This is the same as the command line option */
      /* --vex-guest-chase-thresh=0                  */
   VG_(clo_vex_control).guest_chase_thresh = 0;
}

   /* Parse the command line options */
static Bool bbv_process_cmd_line_option(Char* arg)
{
   if VG_INT_CLO       (arg, "--interval-size",    interval_size) {}
   else if VG_STR_CLO  (arg, "--bb-out-file",      clo_bb_out_file) {}
   else if VG_STR_CLO  (arg, "--pc-out-file",      clo_pc_out_file) {
      generate_pc_file = True;
   }
   else if VG_BOOL_CLO (arg, "--instr-count-only", instr_count_only) {}
   else {
      return False;
   }
```

```
   return True;
}

static void bbv_print_usage(void)
{
   VG_(printf)(
"    --bb-out-file=<file>        filename for BBV info\n"
"    --pc-out-file=<file>        filename for BB addresses and function names\n"
"    --interval-size=<num>       interval size\n"
"    --instr-count-only=yes|no   only print total instruction count\n"
   );
}

static void bbv_print_debug_usage(void)
{
   VG_(printf)("    (none)\n");
}

static void bbv_fini(Int exitcode)
{
   Int i;

   if (generate_pc_file) {
      dumpPcFile();
   }

   for(i=0;i<allocated_threads;i++) {

      if (bbv_thread[i].total_instr!=0) {

         VG_(sprintf)(buf,"\n\n"
                          "# Thread %d\n"
                          "#    Total intervals: %d (Interval Size %d)\n"
                          "#    Total instructions: %lld\n"
                          "#    Total reps: %lld\n"
                          "#    Unique reps: %lld\n"
                          "#    Total fldcw instructions: %lld\n\n",
                 i,
                 (Int)(bbv_thread[i].total_instr/(ULong)interval_size),
                 interval_size,
                 bbv_thread[i].total_instr,
                 bbv_thread[i].global_rep_count,
                 bbv_thread[i].unique_rep_count,
                 bbv_thread[i].fldcw_count);

            /* Print results to display */
         VG_(umsg)("%s\n", buf);

            /* open the output file if it hasn't already */
         if (bbv_thread[i].bbtrace_fd < 0) {
            bbv_thread[i].bbtrace_fd=open_tracefile(i);
         }
            /* Also print to results file */
         VG_(write)(bbv_thread[i].bbtrace_fd,(void*)buf,VG_(strlen)(buf));
```

```
            VG_(close)(bbv_thread[i].bbtrace_fd);
        }
    }
}

static void bbv_pre_clo_init(void)
{
    VG_(details_name)            ("exp-bbv");
    VG_(details_version)         (NULL);
    VG_(details_description)     ("a SimPoint basic block vector generator");
    VG_(details_copyright_author)(
       "Copyright (C) 2006-2009 Vince Weaver");
    VG_(details_bug_reports_to)  (VG_BUGS_TO);

    VG_(basic_tool_funcs)          (bbv_post_clo_init,
                                    bbv_instrument,
                                    bbv_fini);

    VG_(needs_command_line_options)(bbv_process_cmd_line_option,
                                    bbv_print_usage,
                                    bbv_print_debug_usage);

    VG_(track_start_client_code)( bbv_thread_called );


    instr_info_table = VG_(OSetGen_Create)(/*keyOff*/0,
                                           NULL,
                                           VG_(malloc), "bbv.1", VG_(free));

    bbv_thread=allocate_new_thread(bbv_thread,0,allocated_threads);
}

VG_DETERMINE_INTERFACE_VERSION(bbv_pre_clo_init)

/*--------------------------------------------------------------------*/
/*--- end                                                          ---*/
/*--------------------------------------------------------------------*/
```

APPENDIX J

QEMU BBV PATCH CODE LISTING

Here is the BBV generating patch for Qemu, against the git development tree as of 13 January 2010. A few additional patches are required for proper Alpha and MIPS support (hopefully those will be merged soon).

```
diff --git a/exec-all.h b/exec-all.h
index 820b59e..fd22d13 100644
--- a/exec-all.h
+++ b/exec-all.h
@@ -151,6 +151,7 @@ struct TranslationBlock {
     struct TranslationBlock *jmp_next[2];
     struct TranslationBlock *jmp_first;
     uint32_t icount;
+    uint32_t unique_id;
 };

 static inline unsigned int tb_jmp_cache_hash_page(target_ulong pc)
diff --git a/exec.c b/exec.c
index 1190591..1997af1 100644
--- a/exec.c
+++ b/exec.c
@@ -1167,6 +1167,8 @@ static inline void tb_alloc_page(TranslationBlock *tb,
 #endif /* TARGET_HAS_SMC */
 }

+int tb_count=0;
+
 /* Allocate a new translation block. Flush the translation buffer if
    too many translation blocks or too much generated code. */
 TranslationBlock *tb_alloc(target_ulong pc)
@@ -1179,6 +1181,8 @@ TranslationBlock *tb_alloc(target_ulong pc)
     tb = &tbs[nb_tbs++];
     tb->pc = pc;
     tb->cflags = 0;
+    tb->unique_id = tb_count;
+    tb_count++;
     return tb;
 }

diff --git a/linux-user/syscall.c b/linux-user/syscall.c
index 1acf1f5..3b59366 100644
--- a/linux-user/syscall.c
+++ b/linux-user/syscall.c
@@ -86,6 +86,8 @@
 #include "qemu.h"
 #include "qemu-common.h"

+void do_dump_pc(uint32_t);
```

```
+
 #if defined(CONFIG_USE_NPTL)
 #define CLONE_NPTL_FLAGS2 (CLONE_SETTLS | \
     CLONE_PARENT_SETTID | CLONE_CHILD_SETTID | CLONE_CHILD_CLEARTID)
@@ -4194,6 +4196,7 @@ abi_long do_syscall(void *cpu_env, int num, abi_long arg1,
 #ifdef TARGET_GPROF
         _mcleanup();
 #endif
+        do_dump_pc(0xffffffff);
         gdb_exit(cpu_env, arg1);
         _exit(arg1);
         ret = 0; /* avoid warning */
@@ -5718,6 +5721,7 @@ abi_long do_syscall(void *cpu_env, int num, abi_long arg1,
 #ifdef TARGET_GPROF
         _mcleanup();
 #endif
+        do_dump_pc(0xffffffff);
         gdb_exit(cpu_env, arg1);
         ret = get_errno(exit_group(arg1));
         break;
diff --git a/target-alpha/helper.c b/target-alpha/helper.c
index be7d37b..340aadd 100644
--- a/target-alpha/helper.c
+++ b/target-alpha/helper.c
@@ -25,6 +25,8 @@
 #include "exec-all.h"
 #include "softfloat.h"

+#include "../bbv_routines.h"
+
 uint64_t cpu_alpha_load_fpcr (CPUState *env)
 {
     uint64_t ret = 0;
diff --git a/target-alpha/helper.h b/target-alpha/helper.h
index bedd3c0..efc145d 100644
--- a/target-alpha/helper.h
+++ b/target-alpha/helper.h
@@ -1,5 +1,7 @@
 #include "def-helper.h"

+DEF_HELPER_1(dump_pc, void, i32)
+
 DEF_HELPER_2(excp, void, int, int)
 DEF_HELPER_0(load_pcc, i64)
 DEF_HELPER_0(rc, i64)
diff --git a/target-alpha/translate.c b/target-alpha/translate.c
index 87813e7..53a0315 100644
--- a/target-alpha/translate.c
+++ b/target-alpha/translate.c
@@ -2626,6 +2626,15 @@ static inline void gen_intermediate_code_internal(CPUState *env,
         if (num_insns + 1 == max_insns && (tb->cflags & CF_LAST_IO))
             gen_io_start();
         insn = ldl_code(ctx.pc);
+       {
+            /* vmw */
```

```
+             TCGv const1;
+
+             const1 = tcg_const_i32(tb->unique_id);
+             gen_helper_dump_pc(const1);
+             tcg_temp_free(const1);
+         }
+
          num_insns++;

         if (unlikely(qemu_loglevel_mask(CPU_LOG_TB_OP))) {
diff --git a/target-arm/helper.c b/target-arm/helper.c
index b3aec99..caa7549 100644
--- a/target-arm/helper.c
+++ b/target-arm/helper.c
@@ -9,6 +9,8 @@
 #include "qemu-common.h"
 #include "host-utils.h"

+#include "../bbv_routines.h"
+
 static uint32_t cortexa9_cp15_c0_c1[8] =
 { 0x1031, 0x11, 0x000, 0, 0x00100103, 0x20000000, 0x01230000, 0x00002111 };

diff --git a/target-arm/helpers.h b/target-arm/helpers.h
index 0d1bc47..5f58e87 100644
--- a/target-arm/helpers.h
+++ b/target-arm/helpers.h
@@ -1,5 +1,7 @@
 #include "def-helper.h"

+DEF_HELPER_1(dump_pc, void, i32)
+
 DEF_HELPER_1(clz, i32, i32)
 DEF_HELPER_1(sxtb16, i32, i32)
 DEF_HELPER_1(uxtb16, i32, i32)
diff --git a/target-arm/translate.c b/target-arm/translate.c
index 5cf3e06..312d5e6 100644
--- a/target-arm/translate.c
+++ b/target-arm/translate.c
@@ -5964,7 +5964,7 @@ static void gen_store_exclusive(DisasContext *s,
 }
 #endif

-static void disas_arm_insn(CPUState * env, DisasContext *s)
+static void disas_arm_insn(CPUState * env, DisasContext *s, int unique_id)
 {
     unsigned int cond, insn, val, op1, i, shift, rm, rs, rn, rd, sh;
     TCGv tmp;
@@ -5975,7 +5975,16 @@ static void disas_arm_insn(CPUState * env, DisasContext *s)

     insn = ldl_code(s->pc);
     s->pc += 4;
-
+
+    {
```

```
+        /* vmw */
+        TCGv const1;
+
+        const1 = tcg_const_i32(unique_id);
+        gen_helper_dump_pc(const1);
+        tcg_temp_free(const1);
+     }
+
     /* M variants do not implement ARM mode.  */
     if (IS_M(env))
         goto illegal_op;
@@ -9061,7 +9070,7 @@ static inline void gen_intermediate_code_internal(CPUState
                 }
             }
         } else {
-            disas_arm_insn(env, dc);
+            disas_arm_insn(env, dc, tb->unique_id);
         }
         if (num_temps) {
             fprintf(stderr, "Internal resource leak before %08x\n", dc->pc);
diff --git a/target-i386/helper.c b/target-i386/helper.c
index 049fccf..4e4b7b3 100644
--- a/target-i386/helper.c
+++ b/target-i386/helper.c
@@ -30,6 +30,8 @@

 //#define DEBUG_MMU

+#include "../bbv_routines.h"
+
 /* feature flags taken from "Intel Processor Identification and the CPUID
  * Instruction" and AMD's "CPUID Specification". In cases of disagreement
  * about feature names, the Linux name is used. */
diff --git a/target-i386/helper.h b/target-i386/helper.h
index 6b518ad..4a5fa43 100644
--- a/target-i386/helper.h
+++ b/target-i386/helper.h
@@ -1,5 +1,7 @@
 #include "def-helper.h"

+DEF_HELPER_1(dump_pc, void, i32)
+
 DEF_HELPER_FLAGS_1(cc_compute_all, TCG_CALL_PURE, i32, int)
 DEF_HELPER_FLAGS_1(cc_compute_c, TCG_CALL_PURE, i32, int)

diff --git a/target-i386/op_helper.c b/target-i386/op_helper.c
index 5eea322..4d93fa5 100644
--- a/target-i386/op_helper.c
+++ b/target-i386/op_helper.c
@@ -23,7 +23,6 @@

 //#define DEBUG_PCALL

-
 #ifdef DEBUG_PCALL
```

```
 #  define LOG_PCALL(...) qemu_log_mask(CPU_LOG_PCALL, ## __VA_ARGS__)
 #  define LOG_PCALL_STATE(env) \
diff --git a/target-i386/translate.c b/target-i386/translate.c
index 511a4ea..25797c5 100644
--- a/target-i386/translate.c
+++ b/target-i386/translate.c
@@ -4075,7 +4075,8 @@ static void gen_sse(DisasContext *s, int b, target_ulong

 /* convert one instruction. s->is_jmp is set if the translation must
    be stopped. Return the next pc value */
-static target_ulong disas_insn(DisasContext *s, target_ulong pc_start)
+static target_ulong disas_insn(DisasContext *s, target_ulong pc_start,
+                               int unique_id)
 {
     int b, prefixes, aflag, dflag;
     int shift, ot;
@@ -4208,6 +4209,20 @@ static target_ulong disas_insn(DisasContext *s,
     if (prefixes & PREFIX_LOCK)
         gen_helper_lock();

+    {
+       /* vmw */
+       TCGv const1;
+
+       if (prefixes & (PREFIX_REPZ | PREFIX_REPNZ)) {
+         const1 = tcg_const_i32(unique_id|0x80000000);
+       }
+       else {
+         const1 = tcg_const_i32(unique_id);
+       }
+       gen_helper_dump_pc(const1);
+       tcg_temp_free(const1);
+    }
+
     /* now check op code */
  reswitch:
     switch(b) {
@@ -7849,7 +7864,7 @@ static inline void gen_intermediate_code_internal(
         if (num_insns + 1 == max_insns && (tb->cflags & CF_LAST_IO))
             gen_io_start();

-        pc_ptr = disas_insn(dc, pc_ptr);
+        pc_ptr = disas_insn(dc, pc_ptr, tb->unique_id);
         num_insns++;
         /* stop translation if indicated */
         if (dc->is_jmp)
diff --git a/target-mips/helper.c b/target-mips/helper.c
index 903987b..dd0e4f9 100644
--- a/target-mips/helper.c
+++ b/target-mips/helper.c
@@ -34,6 +34,8 @@ enum {
     TLBRET_MATCH = 0
 };

+#include "../bbv_routines.h"
```

```
+
 /* no MMU emulation */
 int no_mmu_map_address (CPUState *env, target_phys_addr_t *physical,
                         target_ulong address, int rw, int access_type)
diff --git a/target-mips/helper.h b/target-mips/helper.h
index ab47b1a..e4faff4 100644
--- a/target-mips/helper.h
+++ b/target-mips/helper.h
@@ -1,5 +1,7 @@
 #include "def-helper.h"

+DEF_HELPER_1(dump_pc, void, i32)
+
 DEF_HELPER_2(raise_exception_err, void, i32, int)
 DEF_HELPER_1(raise_exception, void, i32)
 DEF_HELPER_0(interrupt_restart, void)
diff --git a/target-mips/translate.c b/target-mips/translate.c
index dfea6f6..f700599 100644
--- a/target-mips/translate.c
+++ b/target-mips/translate.c
@@ -9524,9 +9524,17 @@ gen_intermediate_code_internal (CPUState *env,
         if (!(ctx.hflags & MIPS_HFLAG_M16)) {
             ctx.opcode = ldl_code(ctx.pc);
             insn_bytes = 4;
+            {
+                /* vmw */
+                gen_helper_0i(dump_pc,tb->unique_id);
+            }
             decode_opc(env, &ctx, &is_branch);
         } else if (env->insn_flags & ASE_MIPS16) {
             ctx.opcode = lduw_code(ctx.pc);
+            {
+                /* vmw */
+                gen_helper_0i(dump_pc,tb->unique_id);
+            }
             insn_bytes = decode_mips16_opc(env, &ctx, &is_branch);
         } else {
             generate_exception(&ctx, EXCP_RI);
diff --git a/target-ppc/helper.c b/target-ppc/helper.c
index b233d4f..75adac4 100644
--- a/target-ppc/helper.c
+++ b/target-ppc/helper.c
@@ -29,6 +29,8 @@
 #include "qemu-common.h"
 #include "kvm.h"

+#include "../bbv_routines.h"
+
 //#define DEBUG_MMU
 //#define DEBUG_BATS
 //#define DEBUG_SLB
diff --git a/target-ppc/helper.h b/target-ppc/helper.h
index 40d4ced..b34a83e 100644
--- a/target-ppc/helper.h
+++ b/target-ppc/helper.h
```

```
@@ -1,5 +1,7 @@
 #include "def-helper.h"

+DEF_HELPER_1(dump_pc, void, i32)
+
 DEF_HELPER_2(raise_exception_err, void, i32, i32)
 DEF_HELPER_1(raise_exception, void, i32)
 DEF_HELPER_3(tw, void, tl, tl, i32)
diff --git a/target-ppc/translate.c b/target-ppc/translate.c
index d4e81ce..623c045 100644
--- a/target-ppc/translate.c
+++ b/target-ppc/translate.c
@@ -9029,6 +9029,16 @@ static inline void gen_intermediate_code_internal(
         } else {
             ctx.opcode = ldl_code(ctx.nip);
         }
+        {
+          /* vmw */
+          TCGv const1;
+
+          const1 = tcg_const_i32(tb->unique_id);
+          gen_helper_dump_pc(const1);
+
+          tcg_temp_free(const1);
+       }
+
         LOG_DISAS("translate opcode %08x (%02x %02x %02x) (%s)\n",
                   ctx.opcode, opc1(ctx.opcode), opc2(ctx.opcode),
                   opc3(ctx.opcode), little_endian ? "little" : "big");
diff --git a/target-sparc/helper.c b/target-sparc/helper.c
index e801474..9a32974 100644
--- a/target-sparc/helper.c
+++ b/target-sparc/helper.c
@@ -38,6 +38,8 @@ static int cpu_sparc_find_by_name(sparc_def_t *cpu_def,

 static spinlock_t global_cpu_lock = SPIN_LOCK_UNLOCKED;

+#include "../bbv_routines.h"
+
 void cpu_lock(void)
 {
     spin_lock(&global_cpu_lock);
diff --git a/target-sparc/helper.h b/target-sparc/helper.h
index 6f103e7..7e74a21 100644
--- a/target-sparc/helper.h
+++ b/target-sparc/helper.h
@@ -1,5 +1,7 @@
 #include "def-helper.h"

+DEF_HELPER_1(dump_pc, void, i32)
+
 #ifndef TARGET_SPARC64
 DEF_HELPER_0(rett, void)
 DEF_HELPER_1(wrpsr, void, tl)
diff --git a/target-sparc/translate.c b/target-sparc/translate.c
```

```
index 7e9f0cf..58405c5 100644
--- a/target-sparc/translate.c
+++ b/target-sparc/translate.c
@@ -1695,7 +1695,7 @@ static inline void gen_load_trap_state_at_tl(TCGv_ptr
         goto nfpu_insn;

 /* before an instruction, dc->pc must be static */
-static void disas_sparc_insn(DisasContext * dc)
+static void disas_sparc_insn(DisasContext * dc, int unique_id)
 {
     unsigned int insn, opc, rs1, rs2, rd;
     target_long simm;
@@ -1703,6 +1703,16 @@ static void disas_sparc_insn(DisasContext * dc)
     if (unlikely(qemu_loglevel_mask(CPU_LOG_TB_OP)))
         tcg_gen_debug_insn_start(dc->pc);
     insn = ldl_code(dc->pc);
+
+    {
+       /* vmw */
+       TCGv const1;
+
+       const1 = tcg_const_i32(unique_id);
+       gen_helper_dump_pc(const1);
+       tcg_temp_free(const1);
+    }
+
     opc = GET_FIELD(insn, 0, 1);

     rd = GET_FIELD(insn, 2, 6);
@@ -4732,7 +4742,7 @@ static inline void gen_intermediate_code_internal(
         if (num_insns + 1 == max_insns && (tb->cflags & CF_LAST_IO))
             gen_io_start();
         last_pc = dc->pc;
-        disas_sparc_insn(dc);
+        disas_sparc_insn(dc, tb->unique_id);
         num_insns++;

         if (dc->is_br)
diff --git a/bbv_routines.h b/bbv_routines.h
new file mode 100644
index 0000000..16f31d0
--- /dev/null
+++ b/bbv_routines.h
@@ -0,0 +1,101 @@
+/* vmw */
+
+void do_dump_pc(uint32_t bb);
+
+#if !defined(TARGET_ARM)
+void helper_dump_pc(uint32_t bb);
+#endif
+
+void gen_helper_dump_pc(uint32_t bb);
+
+
```

```
+#define MAX_BBS 100000
+#define INTERVAL_SIZE 100000000 /* 100 million */
+
+void do_dump_pc(unsigned int bb) {
+
+   static unsigned long total_count=0,intervals=0;
+   static int bbvs[MAX_BBS];
+   int i;
+   static FILE *bbv_file=NULL;
+
+#if defined(TARGET_I386) || defined (TARGET_X86_64)
+   static int rep_count=0;
+   int rep;
+   static long long total_reps=0;
+#endif
+
+   if (bb==0xffffffff) {
+      if (bbv_file!=NULL) {
+        long long total;
+        total=((long long)intervals*INTERVAL_SIZE)+(long long)total_count;
+        fprintf(bbv_file,"# Total count : %lld\n",total);
+#if defined(TARGET_I386) || defined (TARGET_X86_64)
+        fprintf(bbv_file,"# Rep count    : %lld\n",total_reps);
+#endif
+        fclose(bbv_file);
+        return;
+      }
+   }
+
+   if (bbv_file==NULL) {
+      bbv_file=fopen("qemusim.bbv","w");
+      if (bbv_file==NULL) {
+        printf("Error!  Could not open file %s\n","qemusim.bbv");
+        exit(-1);
+      }
+   }
+
+#if defined(TARGET_I386) || defined (TARGET_X86_64)
+   rep=bb&0x80000000;
+   bb  &=0x7fffffff;
+#endif
+
+   if (bb>MAX_BBS) {
+      printf("Error!  Not enough BBS %d\n",bb);
+      exit(-1);
+   }
+
+#if defined(TARGET_I386) || defined (TARGET_X86_64)
+
+   if (rep) {
+      rep_count++;
+      total_reps++;
+      return;
+   }
+
```

```
+   if ((rep_count) && (!rep)) {
+      rep_count=0;
+      /* count all reps as one instruction (as per docs) */
+      /* this makes things match perf-ctr results        */
+      total_count++;
+      bbvs[bb]++;
+   }
+#endif
+
+   total_count++;
+   bbvs[bb]++;
+
+   if (total_count>=INTERVAL_SIZE) {
+      intervals++;
+      fprintf(bbv_file,"T");
+      for(i=0;i<MAX_BBS;i++) {
+        if (bbvs[i]) {
+           /* simpoint can't handle a basic block starting at zero? */
+           fprintf(bbv_file,":%d:%d ",i+1,bbvs[i]);
+        }
+      }
+      fprintf(bbv_file,"\n");
+
+      /* clear the stats */
+      total_count=0;
+      for(i=0;i<MAX_BBS;i++) {
+        bbvs[i]=0;
+      }
+   }
+}
+
+/* grrr, why is this needed on x86 */
+void helper_dump_pc(unsigned int bb) {
+   do_dump_pc(bb);
+}
```

APPENDIX K

R12000 BRANCH PREDICTOR KERNEL MODULE

This is patch against the MIPS Linux kernel allows setting the branch predictor behavior on an R12000 processor, as described in Chapter 5.

```
/*
 *  brpred_config.c  -  configure branch predictor on R12000
 *  TODO - configurable by module parameter, not by
 *         recompiling
 */
#include <linux/module.h>       /* Needed by all modules */
#include <linux/kernel.h>       /* Needed for KERN_INFO */
#include <asm/mipsregs.h>

int init_module(void) {

  unsigned int x;

  x=__read_32bit_c0_register($22, 0);

  printk(KERN_INFO "Hello world. %x\n",x);

  /*
   * A non 0 return means init_module failed; module can't be loaded.
   */

   __write_32bit_c0_register($22, 0, 0x20300000);  /* default 2-bit */
// __write_32bit_c0_register(£22, 0, 0x20310000);  /* not-taken */
// __write_32bit_c0_register(£22, 0, 0x20320000);  /* taken */
// __write_32bit_c0_register(£22, 0, 0x20330000);  /* fwd=not, back=yes */


  return 0;
}

void cleanup_module(void) {

   unsigned int x;

   x=__read_32bit_c0_register($22, 0);
   printk(KERN_INFO "Goodbye world %x.\n",x);
}
```

APPENDIX L

SESC R12000 CONFIGURATION FILE

We use this SESC configuration file to model an R12000 processor with 2-bit branch prediction (as described in Chapter 5).

```
############################
# General Processor Options
############################

nCPUs     = 1           # We have a single core
cpucore[0:0] = 'issueX' # single core

# Parameters
procsPerNode = 1        # our machine is single processor
pageSize     = 4096     # under Linux at least


#################
# Technology
#################
technology  = 'techParam'

[techParam]
tech        = 250               # nm
frequency   = 300e6             # Hz


###############################
#  PROCESSOR CONFIGURATION    #
###############################

# r12k p 13
# int register file
# 3 write ports, 7 read ports
#   each ALU has two read and 1 write
#   addr calc unit has 2 read ports
#   last  read shared between store, jr, and move-to-fp
#   last write shared between load, bal, and move-from-fp
# no rob
# special "condition" file for conditional move instructions

# fp register file
# 3 write ports, 5 read ports
# + adder and mul each has 2 dedicated read and one write
# last read is shared between store and move


[issueX]
issueWrongPath = true   # only if compiled with SESC_MISPATH
inorder        = false  # r12k paper page 1
```

```
fetchWidth      = 4       # r12k paper page 1
instQueueSize   = 4       # Renau - This is a different type of structure
                          # !(r10k paper page 33)
issueWidth      = 4   # Renau - By issue SESC means max rename per cycle
                      # !(from r12k paper, page 1)
retireWidth     = 4   # r10k paper page 33
decodeDelay     = 1   # Renau
renameDelay     = 1   # Renau
maxBranches     = 4   # does this mean max branches we can run through? r12k
bb4Cycle        = 1   # ??
maxIRequests    = 3   # ??
interClusterLat= 1    # Renau
cluster[0]      = 'FXClusterIssueX'
cluster[1]      = 'FPClusterIssueX'
cluster[2]      = 'AddressIssueX' # r12k has separate address queue
stForwardDelay = 1    # ??
maxLoads        = 16 # Renau - ld/st share a single queue with 16 entries
maxStores       = 16 # Renau
regFileDelay    = 1   # Renau
robSize         = 48 # ?? r12k does not have a ROB but has a 48-entry active list (p9)
intRegs         = 64 # r12k p8
fpRegs          = 64 # r12k p8
bpred           = 'BPredIssueX'
dataSource      = "DataL1 DL1"
instrSource     = "InstL1 IL1"
enableICache    = true
dtlb            = 'TLB'
itlb            = 'TLB'
OSType          = 'std'

minTLBMissDelay= 50 # ?? Linux reports around 50 instructions for TLB miss
                    # handler


########################
# TLB
########################
# r12k has a 64-entry unified TLB
#  each entry points to two consecutive pages
#  it is fully associative
#  it is not possible to represent this in SESC?
# replacement is done in software.  Often the bottom 8 entries
#  are fixed and the rest is random?

# more info, r12kpaper p16

# also, typically 8 of the entries are pinned to the OS

[TLB]
deviceType='tlb'
size =  64*8            # is this bytes?
assoc = 64              # we want fully-associative
bsize = 8               # block size???
numPorts = 1            # have no idea
replPolicy = 'LRU'      # ??
```

```
#
# Pipeline clusters
#

# Int ALU
#
# Either ALU can add/sub/logical/move hilo/trap
#  ALU1 branches, shift, lui, conditional moves
#  ALU2 mul,div
# Load/Store unit


[FXClusterIssueX]
blockName    = "IntWin"
winSize      = 16          #??
recycleAt    = 'Execute'   # Recycle entries at : Execute|Retire
                           # it looks like Execute is right for r12k
schedNumPorts = 2  # Renau - 2 execution units max
schedPortOccp = 1  # ??
wakeUpNumPorts= 0  # Renau - No split wakeup/select cycle in R12k
wakeUpPortOccp= 0  # Renau
wakeupDelay   = 0  # Renau
schedDelay    = 0  # Renau


iALUUnit     = 'ALUIssueX'
iALULat      = 1   # r12k paper, p13

iBJUnit      = 'ALUIssueX'      # Branch jump?
iBJLat       = 1   # r12k paper, p13

iDivUnit     = 'MDIssueX'
iDivLat      = 35        # r12k, 34/35 cycles for 32 bit, 66/67 for 64 bit
                         # r12k paper, p13

iMultUnit    = 'MDIssueX'
iMultLat     = 6         # r12k, 5/6 for 32-bit, unsigned is 1 extra
                         # 9/10 for 64-bit
                         # r12k paper, p13

[AddressIssueX]
blockName    = "IntWin"
winSize      = 16          #??
recycleAt    = 'Execute'   # Recycle entries at : Execute|Retire

schedNumPorts = 2  # ??
schedPortOccp = 1  # ??
wakeUpNumPorts= 0  # ??
wakeUpPortOccp= 0  # ??
wakeupDelay   = 0  # ??
schedDelay    = 0  # ??
```

```
iStoreUnit   = 'LDSTIssueX' # Renau - shared LD/ST cache port
iStoreLat    = 1   # ??

iLoadUnit    = 'LDSTIssueX'
iLoadLat     = 1   # r12k paper, p13 if in l1 cache.
                   # is this cumulative with cache hit rate?




#
# FP ALU
#
# For r12k, there are 5 floating point units
#  they are 3-stage pipelined, with a 1-cycle repeat rate
# adder
# multiplier
# divide
# square root
# load/store
# latency and repeat rate are not necessarily the same

[FPClusterIssueX]
blockName    = "FPWin"
winSize      = 16        # ??
recycleAt    = 'Execute' # Recycle entries at : Execute|Retire
schedNumPorts = 2 # Renau
schedPortOccp = 1 # ??
wakeUpNumPorts= 0 # Renau
wakeUpPortOccp= 0 # Renau
wakeupDelay   = 0 # Renau
schedDelay    = 0 # ??

fpALUUnit    = 'FP0IssueX'
fpALULat     = 2  # r12k paper p16

fpMultUnit   = 'FP1IssueX'  # Renau
fpMultLat    = 2  # 4 if it is a multiply/add
                  # r12k p16

fpDivUnit    = 'FP1IssueX' # Renau
fpDivLat     = 12 # 12 for 32bit, 19 for 64-bit
                  # r12k p16

[LDSTIssueX]
Num = 1 # ??
Occ = 1 # ??

[ALUIssueX]
Num = 1 # Renau
Occ = 1 # ??

[MDIssueX] # Renau - muldiv for int
Num = 1
```

```
Occ = 8 # Renau - Shared by mult and div.  32b mult 6, 64b mult 10,
        # 32 bit div 35, 64b div 67.
        # What is the real mix?  Not easy to model with sesc

[FP0IssueX]
Num = 1 # ??
Occ = 1 # ??

[FP1IssueX] # Renau - FP Mult/Div unit
Num = 1
Occ = 2  # Renau - 2 cycle occupancy

########################################
# BRANCH PREDICTOR
########################################
# bits 12:2 used to index into 2048 entry saturating
# 2-bit counter
#
# 4 "shadow" copies of reg file.  When misprediction,
# takes 2 cycles to recover on r12k (1 cycle on r10k)
#
# r12k has optional 8-bit global history that can be hashed
#   into main branch index. Linux leaves this alone, disabled
#   by default (r12k p19)


[BPredIssueX]
type           = "2bit"
size           = 2048      # r12k paper, p5
rasSize        = 4         # r12k manual page 236
btbSize        = 32        # r12k paper p5
btbAssoc       = 2         # r12k paper p5
btbBsize       = 1         # ??
btbReplPolicy  = 'LRU'     # ??
bits           = 2         # (saturating bits.  why is this not
                           #  documented better)
BTACDelay      = 2         # Renau

###############################
# physically tagged
# 32kB 2-way 64-byte lines
# from dmesg on actual machine
# also has a parity bit
# stored in a 36-bit pre-decoded format
# LRU
# Can fetch 4 consecutive instructions, but cannot cross a block boundary

[InstL1]
deviceType  = 'icache'
blockName   = "Icache"
size        = 32*1024
assoc       = 2
bsize       = 64
writePolicy = 'WB'  # n/a
replPolicy  = 'LRU' #
```

```
numPorts     = 1      # Renau
portOccp     = 1      #??
hitDelay     = 2      # cycles? if so, r12k p6
missDelay    = 0      #??
MSHR         = 'InstL1MSHR' #??
lowerLevel   =  "L2Cache L2 shared"

[InstL1MSHR]
size = 4         # Renau
                 # ?? is this bytes? entries?
type = 'full'     # ??
bsize = 64        # ??

################################
# Memory Subsystem (L1)
################################
#  32kB, 2-way, linesize 32 bytes
# each byte has parity bit
# two identical banks selected by address bit 5
# physically tagged

[DataL1]
deviceType   = 'cache'
blockName    = "Dcache"
MSHR         = "DL1MSHR"
size         = 32*1024
assoc        = 2
skew         = false  #??
bsize        = 32
replPolicy   = 'LRU' #
numPorts     =  2 #Renau
portOccp     =  1 #??
hitDelay     =  2 # cycles? if so r12k p6
                  # lmbench shows 6.6ns, so that matches 2 cycles

missDelay    =  1 #??
writePolicy  = "WB" # r12k p7
lowerLevel   = "CommonBus Bus shared" #??

[DL1MSHR]
type = 'full' #??
size = 8      # Renau - 4MSHR entries in R10K
              #         but can handle up to 4 pending per entry
              #         SESC cannot do this, approx with 8?
bsize = 64    #??


##############################
# 2MB, 2-way, 128 bytes
# from dmesg
# each quadword has 9-bit ECC and a parity bit
# correction pipeline takes 2 cycles
# latency ~10 cycles

# has a 16kB way-predict table
```

```
# the tags re on-chip, cache itself off-chip

[L2Cache]
deviceType    = 'cache'
blockName     = "L2"
size          = 2*1024*1024
assoc         = 2
bsize         = 128
writePolicy   ='WB'  # r12k p7
replPolicy    ='LRU' #??
numPorts      =  1   #??
portOccp      =  1   #??
hitDelay      = 10   # r12k p 6
                     # lmbench measurements show 47.3ns or about 14cycles


missDelay     =  4   #??
MSHR          = 'MSHRL2' #??
lowerLevel    = "MemoryBus"

[MSHRL2]
type = 'full' # ??
size = 32     # ??
bsize = 64    # ??

[CommonBus]
deviceType   = 'bus'
busWidth     =      32
busLength    =    7500  # 7.5 mm ??
numPorts     =       1  # Renau
portOccp     =       1
delay        =       1  # Renau
buffWCReqs   =       1
lowerLevel   =  "L2Cache L2 shared"


[MemoryBus]
deviceType    = 'bus'
numPorts      = 1
portOccp      = 168    # Renau - Since the processor operates at 300MHz and
                       # the L2 cache has 128bytes, to have around 300MB/s
                       # when you just need 1 request every 128 cyles. To
                       # make it closer to 220MB/s use 168 cycles port
                       # occupancy (128/168*300 = 220MB/s)


                       # lmbench shows ~220MB/s bandwidth
                       #   220MB/s / 128B = 1.802e6 cachelines/s
                       #                  = .005946 cachelines/cycle
                       # so, 168 cycles/cacheline?

delay         = 15
lowerLevel    = "Memory Memory"
```

```
# The Octane we have has a peak bandwidth 1.0GB/s system bus
# 2GB of SDRAM memory, possibly PC100 (100MHz)

[Memory]
deviceType     = 'niceCache'
size           = 128
assoc          = 1
bsize          = 64
writePolicy    = 'WB'
replPolicy     = 'LRU'
numPorts       = 1
portOccp       = 1
hitDelay       = 113    # According to lmbench our machine has 404.6ns delay
                        # clock cycle is 300MHz so 3.3ns
                        # so approximately 120 clock cycles
                        # renau - You have to discount the miss time
                        # (21 cycles total), so 113 clk should be
                        # fine (404/3-21 ~ 113).


missDelay      = 500
MSHR           = NoMSHR
lowerLevel     = 'voidDevice'

[NoMSHR]
type = 'none'
size = 128
bsize = 64

[voidDevice]
deviceType     = 'void'
```

BIBLIOGRAPHY

[1] Chart Get!: Media create sales: 04/06 - 04/12. http://chartget.com/ 2009/04/ media-create-sales-0406-0412-hardware.html.

[2] IBM research SimOS website. http://www.research.ibm.com/ arl/projects/SimOSppc.html.

[3] QEMU BBV website. http://www.csl.cornell.edu/ vince/projects/qemusim/.

[4] Snapshot of the embedded Linux market – may, 2006. http://www.linuxfordevices.com/ c/a/Linux-For-Devices-Articles/ Snapshot-of-the-embedded-Linux-market-May-2006/.

[5] Top 500 supercomputing sites. http://www.top500.org/.

[6] Advanced Micro Devices. *AMD Athlon Processor Model 6 Revision Guide*, 2003.

[7] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual*, 2006.

[8] A.R. Alameldeen and D.A. Wood. Variability in architectural simulations of multi-threaded commercial workloads. In *Proc. 9th IEEE Symposium on High Performance Computer Architecture*, 2003.

[9] N.M. Amato, J. Perdue, M.M. Mathis, A. Pietracaprina, and G. Pucci. Predicting performance on smps. a case study: The SGI power challence. In *Proc. 14th IEEE/ACM International Parallel and Distributed Processing Symposium*, page 729, 2000.

[10] AMD. *AMD Family 10h Processor BIOS and Kernel Developer Guide*, 2009.

[11] ARM Limited. *ARM Architecture Reference Manual*, 2000.

[12] Atmel. *AVR32 Architecture Document*, 2006.

[13] T. Austin. SimpleScalar 4.0 release note. http://www.simplescalar.com/.

[14] Axis Communications AB. *ETRAX FS Designer's Reference*, 2007.

[15] R. Balasubramonian, D.H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proc. IEEE/ACM 33nd International Symposium on Microarchitecture*, pages 245–257, December 2000.

[16] L. Barroso, K. Gharachoroloo, and E. Bugnion. Memory system characterization of commerical workloads. In *Proc. 25th IEEE/ACM International Symposium on Computer Architecture*, June 1998.

[17] R.C. Bedicheck. Talisman: Fast and accurate multicomputer simulation. In *Proc.ACM International Conference on Measurement and Modeling of Computer Systems*, May 1995.

[18] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proc. USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, April 2005.

[19] L. Benini, A. Macii, and A. Nannarelli. Cached-code compression for energy minimization in embedded processors. In *Proc. IEEE/ACM International Symposium on Low Power Electronics and Design*, pages 322–327, August 2001.

[20] Á. Beszédes, R. Ferenc, T. Gyimóthy, A. Dolenc, and K. Karsisto. Survey of code-size reduction methods. *ACM Computing Surveys*, 35(3):223–267, September 2003.

[21] R. Bhargava, L.K. John, and F. Matus. Accurately modeling speculative instruction fetching in trace-driven simulation. pages 65–71, 1999.

[22] N.L. Binkert, R.G. Dreslinski, L.R. Hsu, K.T. Lim, A.G. Saidi, and S.K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[23] R. Bitirgen, E. İpek, and J.F. Martínez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proc. IEEE/ACM 41st Annual International Symposium on Microarchitecture*, pages 318–329, December 2008.

[24] B. Black, A. Huang, M. Lipasti, and J. Shen. Can trace-driven simulators accurately predict superscalar performance? In *Proc. IEEE International Conference on Computer Design*, pages 478–485, October 1996.

[25] B. Black and J. P. Shen. Calibration of microprocessor performance models. *IEEE Computer*, 31(5):59–65, May 1998.

[26] C. Blundell, M.M.K. Martin, and T.F. Wenisch. InvisiFence: Performance transparent memory ordering in conventional multiprocessors. In *Proc. 36th IEEE/ACM International Symposium on Computer Architecture*, pages 233–244, June 2009.

[27] T. Bonny and J. Henkel. Efficient code density through look-up table compression. In *Proc. ACM/IEEE Design, Automation and Test in Europe Conference and Exposition*, pages 809–814, April 2007.

[28] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proc. 27th IEEE/ACM International Symposium on Computer Architecture*, pages 83–94, June 2000.

[29] B.R. Buck and J.K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[30] D. Burger and T.M. Austin. The SimpleScalar toolset, version 2.0. Technical Report 1342, University of Wisconsin, June 1997.

[31] H. Cain, K. Lepak, B. Schwartz, and M. Lipasti. Precise and accurate processor simulation. In *Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 13–22, February 2002.

[32] D. Chiou, D. Sunwoo, H. Angepat, J. Kim, N.A. Patil, W. Reinhart, and D.E. Johnson. Parallelizing computer system simulators. In *Proc. 22nd IEEE/ACM International Parallel and Distributed Processing Symposium*, pages 1–5, April 2008.

[33] D. Chiou, D. Sunwoo, J. Kim, N.A. Patil, W. Reinhart, D.E. Johnson, J. Keefe, and H. Angepat. FPGA-accelerated simulation technologies (FAST): Fast, full-system, cycle-accurate simulators. In *Proc. IEEE/ACM 40th Annual International Symposium on Microarchitecture*, pages 249–261, December 2007.

[34] P. Chow and M. Horowitz. Architectural tradeoffs in the design of MIPS-X. In *Proc. 14th IEEE/ACM International Symposium on Computer Architecture*, pages 300–308, June 1987.

[35] D. Citron. MisSPECulation: Partial and misleading use of SPEC CPU2000 in computer architecture conferences. In *Proc. 30th IEEE/ACM International Symposium on Computer Architecture*, pages 52–62, June 2003.

[36] Compaq Computer Corporation. *Alpha Architecture Handbook*, 1998.

[37] G. Contreras, M. Martonosi, J. Peng, R. Ju, and G. Lueh. XTREM: A power simulator for the intel XScale core. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 115–125, 2004.

[38] J.W. Davidson and R.A. Vaughan. The effect of instruction set complexity on program size and memory performance. In *Proc. 2nd ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 60–64, October 1987.

[39] B. De Sutter, B. De Bus, K. De Bosschere, and S. Debray. Combining global code and data compaction. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 29–38, 2001.

[40] L.A. DeRose. The hardware performance monitor toolkit. In *Proc. 7th International Euro-Par Conference*, pages 122–132, August 2001.

[41] L.A. DeRose, K. Ekanadham, J.K. Hollingsworth, and S. Sbaraglia. SIGMA: A simulator infrastructure to guide memory analysis. In *Proc. IEEE/ACM Supercomputing International Conference on High Performance Computing, Networking, Storage and Analysis*, number 6, November 2002.

[42] R. Desikan, D. Burger, and S. Keckler. Measuring experimental error in multiprocessor simulation. In *Proc. 28th IEEE/ACM International Symposium on Computer Architecture*, pages 266–277, June 2001.

[43] R. Desikan, D. Burger, S. Keckler, and T. Austin. Sim-alpha: a validated, execution-driven Alpha 21264 simulator. Technical Report TR-01-23, Department of Computer Sciences, The University of Texas at Austin, 2001.

[44] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared memory multiprocessing. In *Proc. 14th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, 2009.

[45] Digital Equipment Corp. *pdp11/40 Processor Handbook*, 1972.

[46] Digital Equipment Corp. *VAX Architecture Reference Manual*, 1987.

[47] J. Donald and M. Martonosi. An efficient, practical parallelization methodology for multicore architecture simulation. *Computer Architecture Letters*, August 2006.

[48] J. Edler and M.D. Hill. Dinero IV trace-driven uniprocessor cache simulator. http://www.cs.wisc.edu/ markhill/DineroIV, 2003.

[49] L. Eeckhout, A. Georges, and K. De Bosschere. Selecting a reduced but representative workload. In *OOPSLA 2003 Workshop on Middleware Benchmarking: Approaches, Results and Experiences*, 2003.

[50] M. Ekman and P. Stenstrom. Enhancing multiprocessor architecture simulation speed using matched-pair comparison. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, 2005.

[51] S. Eranian. Perfmon2: a flexible performance monitoring interface for Linux. In *Proc. 2006 Ottawa Linux Symposium*, pages 269–288, July 2006.

[52] S. Eyerman, L. Eeckhout, T. Karkhanis, and J.E. Smith. A performance counter architecture for computing accurate CPI components. In *Proc. 12th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 175–184, 2006.

[53] M. Ferdman, T.F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Temporal instruction fetch streaming. In *Proc. IEEE/ACM 41st Annual International Symposium on Microarchitecture*, December 2008.

[54] M.J. Flynn, C.L. Mitchell, and J.M. Mulder. And now a case for more complex instruction sets. *IEEE Computer*, 20(9):71–83, September 1987.

[55] K. Ganesan, D. Panwar, and L.K. John. Generalization, validation and analysis of spec cpu2006 simulation points based on branch, memory and TLB characteristics. In *SPEC Benchmark Workshop*, January 2009.

[56] J. Gibson, R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, and M. Heinrich. FLASH vs. (simulated) FLASH: Closing the simulation loop. In *Proc. 9th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 49–58, November 2000.

[57] S.R. Goldschmidt and J.L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. In *Proc.ACM International Conference on Measurement and Modeling of Computer Systems*, pages 146–157, May 1993.

[58] J. Gonzalez, J. Gimenez, and J. Labarta. Automatic detection of parallel applications computation phases. In *Proc. 23rd IEEE/ACM International Parallel and Distributed Processing Symposium*, pages 1–11, May 2009.

[59] T. Granlund and L. Montgomery. Division by invariant integers using multiplication. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 61–72, June 1994.

[60] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. IEEE 4th Workshop on Workload Characterization*, pages 3–14, December 2001.

[61] A. Halambi, A. Shrivastava, P. Biswas, N. Dutt, and A. Nicolau. A design space exploration framework for reduced bit-width instruction set architecture (rISA) design. In *Proc. 15th IEEE/ACM International Symposium on System Synthesis*, pages 120–125, November 2002.

[62] G. Hamerly, E. Perelman, J. Lau, and B. Calder. SimPoint 3.0: Faster and more flexible program analysis. In *Workshop on Modeling, Benchmarking and Simulation*, June 2005.

[63] A. Hasegawa, I. Kawasaki, K. Yamada, S. Yoshioka, S. Kawasaki, and P. Biswas. SH3: High code density, low power. *IEEE Micro*, 15(6):11–19, 1995.

[64] M. Hauswirth, A. Diwan, P.F. Sweeney, and M.C. Mozer. Automating vertical profiling. In *Proc. 20th ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 281–296, 2005.

[65] K. Hazelwood, G. Lueck, and R. Cohn. Scalable support for multithreaded applications on dynamic binary instrumentation systems. In *Proc. International Symposium on Memory Management*, June 2009.

[66] Z. Herczeg, Á. Kiss, D. Schmidt, N. Wehn, and T. Gyimóthy. Xeemu: An improved xscale power simulator. In *PATMOS*, pages 300–309, 2007.

[67] Hewlett Packard. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 1994.

[68] K. Hoste. Personal communication, 2009.

[69] IBM. *Enterprise Systems Architecture/390: Principles of Operation*, 1999.

[70] IBM. *PowerPC Microprocessor Family: The Programming Environments for 32-bit Microprocessors*, 2000.

[71] Intel. *Intel Itanium Architecture Software Developer's Manual*, 2000.

[72] Intel. *Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*, 2009.

[73] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2007.

[74] A. Jaleel, R. Cohn, C.-K. Luk, and B. Jacob. CMP$im: A binary instrumentation approach to modeling memory behavior of workloads on CMPs. Technical Report UMD-SCA-2006-01, University of Maryland, 2006.

[75] A. Jaleel, R.S. Cohn, C.-K. Luk, and B. Jacob. CMP$im: A Pin-based on-the-fly multi-core cache simulator. In *Proc. Workshop on Modeling, Benchmarking, and Simulation*, pages 28–36, June 2008.

[76] K. Keeton, D. Patterson, Y. He, R. Raphael, and W. Baker. Performance characterization of a quad pentium pro SMP using OLTP workloads. In *Proc. 28th IEEE/ACM International Symposium on Computer Architecture*, June 2001.

[77] AJ KleinOsowski and D.J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, June 2002.

[78] W. Korn, P.J. Teller, and G. Castillo. Just how accurate are performance counters? In *20th IEEE International Performance, Computing, and Communication Conference*, pages 303–310, April 2001.

[79] M. Kozuch and A. Wolfe. Compression of embedded system programs. In *Proc. IEEE International Conference on Computer Design*, pages 270–277, October 1994.

[80] J. Lau, S. Schoenmackers, T. Sherwood, and B. Calder. Reducing code size with echo instructions. In *Proc. 7th ACM International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 84–94, October 2003.

[81] B.C. Lee, J. Collins, H. Wang, and D. Brooks. CPR: Composable performance regression for scalable multiprocessor models. In *Proc. IEEE/ACM 41st Annual International Symposium on Microarchitecture*, pages 270–281, December 2008.

[82] H. Lekatsas and W. Wolf. SAMC: A code compression algorithm for embedded processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(12):1689–1701, 1999.

[83] K.M. Lepak, H.W. Cain, and M.H. Lipasti. Redeeming IPC as a performance metric for multithreaded programs. In *Proc. IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques*, page 232, 2003.

[84] Y. Li, B. Lee, D. Brooks, Z. Hu, and K. Skadron. CMP design space exploration subject to physical constraints. In *Proc. 12th IEEE Symposium on High Performance Computer Architecture*, pages 15–26, February 2006.

[85] C.H. Lin, Y. Xie, and W. Wolf. LZW-based code compression for VLIW embedded systems. In *Proc. ACM/IEEE Design, Automation and Test in Europe Conference and Exposition*, pages 76–81, February 2004.

[86] G.H. Loh, S. Subramaniam, and Y. Xie. Zesto: A cycle-level simulator for highly detailed microarchitecture exploration. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, April 2009.

[87] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, June 2005.

[88] Y. Luo, O.M. Lubeck, H. Wasserman, F. Bassetti, and K.W. Cameron. Development and validation of a hiearchical memory model incorporating cpu- and memory-operation overlap model. In *Workshop on Software Performance*, pages 152–163, 1998.

[89] Y. Luo, V. Packirisamy, W.-C. Hsu, A. Zhai, N. Mungre, and A. Tarkas. Dynamic performance tuning for speculative threads. In *Proc. 36th IEEE/ACM International Symposium on Computer Architecture*, pages 462–473, June 2009.

[90] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proc.ACM International Conference on Measurement and Modeling of Computer Systems*, pages 2–13, June 2004.

[91] M.M.K. Martin, D.J. Sorin, B.M. Beckmann, M.R. Marty, M. Xu, A.R. Alameldeen, K.E. Moore, M.D. Hill, and D.A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, 2005.

[92] J.R. Mashey. War of the benchmark means: Time for a truce. *ACM SIGARCH Computer Architecture News*, 32:1–14, September 2004.

[93] H. Massalin. Superoptimizer: a look at the smallest program. In *Proc. 2nd ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 122–126, October 1987.

[94] W. Mathur and J. Cook. Improved estimation for software multiplexing of performance counting. In *Proc. 13th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 23–34, September 2005.

[95] M.E. Maxwell, P.J. Teller, L.Salayandia, and S. Moore. Accuracy of performance monitoring hardware. In *Proc. Los Alamos Computer Science Institute Symposium*, October 2002.

[96] MIPS Technologies, Inc. *MIPS32 Architecture for Programmers*, 2001.

[97] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proc. 35th IEEE/ACM International Symposium on Computer Architecture*, pages 289–300, June 2008.

[98] P. Montesinos, M. Hicks, S.T. King, and J. Torrellas. Capo: A software-hardware interface for practical deterministic multiprocessor replay. In *Proc. 14th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, March 2009.

[99] MOS Technology Inc. *MCS6500 Microcomputer Family Hardware Manual*, 1975.

[100] Motorola, Inc. *Motorola MC88110 User's Manual*, 1991.

[101] Motorola, Inc. *Motorola M68000 Family Programmer's Reference Manual*, 1992.

[102] P. J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A portable interface to hardware performance counters. In *Proc. Department of Defense HPCMP User Group Conference*, June 1999.

[103] A. Muzahid, D. Suaárez, S. Qi, and J. Torrellas. SigRace: Signature-based data race detection. In *Proc. 36th IEEE/ACM International Symposium on Computer Architecture*, pages 337–348, June 2009.

[104] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney. We have it easy, but do we have it right? In *NSF Next Generation Systems Workshop*, pages 1–5, April 2008.

[105] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney. Producin wrong data without doing anything obviously wrong! In *Proc. 14th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, March 2009.

[106] T. Mytkowicz, P.F. Sweeney, M. Hauswirth, and A. Diwan. Time interpolation: So many metrics, so few registers. In *Proc. IEEE/ACM 41st Annual International Symposium on Microarchitecture*, pages 286–300, 2007.

[107] P. Nagpurkar and C. Krintz. Visualization and analysis of phased behavior in Java programs. In *Proc. ACM 3rd international symposium on Principles and practice of programming in Java*, pages 27–33, June 2004.

[108] A.A. Nair and L.K. John. Simulation points for spec cpu 2006. In *Proc. IEEE International Conference on Computer Design*, pages 397–403, 2008.

[109] J. Namkung, D. Kim, R. Gupta, I Kozintsev, J.-Y. Bouget, and C. Dulong. Phase guided sampling for efficient parallel application simulation. In *Proc. 4th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 187–192, 2006.

[110] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *Proc.ACM International Conference on Measurement and Modeling of Computer Systems*, pages 216–227, 2006.

[111] NEC. *VR10000 Series 64-/32-bit Microprocessor User's Manual*, 2001.

[112] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, 2004.

[113] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, June 2007.

[114] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *Proc. 14th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, March 2009.

[115] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *Proc. IEEE/ACM 37th Annual International Symposium on Microarchitecture*, pages 81–93, December 2004.

[116] D.A. Penry, D.L. August, and M. Vachharajani. Rapid development of a flexible validated processor model. In *Proc. Workshop on Modeling, Benchmarking, and Simulation*, pages 21–30, June 2005.

[117] C. Pereira, H. Patil, and B. Calder. Reproducible simulation of multithreaded workloads for architectural design exploration. In *Proc. IEEE International Symposium on Workload Characterization*, pages 173–182, September 2008.

[118] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *Proc. IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 244–256, September 2003.

[119] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder. Using SimPoint for accurate and efficient simulation. In *Proc.ACM International Conference on Measurement and Modeling of Computer Systems*, pages 318–319, June 2003.

[120] E. Perelman, J. Lau, H. Patil, A. Jaleel, G. Hamerly, and B. Calder. Cross binary simulation points. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, 2007.

[121] E. Perelman, M. Polito, J.-Y. Bouguet, J. Sampson, B. Calder, and C. Dulong. Detecting phases in parallel applications on shared memory archi-

tectures. In *Proc. 20th IEEE/ACM International Parallel and Distributed Processing Symposium*, 2006.

[122] A. Phansalkar, A. Joshi, and L.K. John. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. In *Proc. 34th IEEE/ACM International Symposium on Computer Architecture*, pages 412–413, June 2007.

[123] R. Phelan. *Improving ARM Code Density and Performance: New Thumb Extensions to the ARM Architecture*. ARM Limited, 2003.

[124] B. Raiter. http://www.muppetlabs.com/˜breadbox/software/ elfkickers.html, 2007.

[125] J. Renau. SESC. http://sesc.sourceforge.net/index.html, 2002.

[126] Renesas Technology. *SH-3/SH-3E/SH3-DSP Software Manual*, 2006.

[127] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, 1997.

[128] M. Rosenblum, E. Bugnion, S.A. Jerrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. In *Proc. 15th ACM Symposium on Operating Systems Principles*, 1995.

[129] S. Sarangi, B. Greskamp, A. Tiwari, and J. Torrellas. EVAL: Utilizing processors with variation-induced timing errors. In *Proc. IEEE/ACM 41st Annual International Symposium on Microarchitecture*, pages 423–434, December 2008.

[130] S. Seong and P. Mishra. A bitmask-based code compression technique for embedded systems. In *Proc. International Conference on Computer Aided Design*, pages 251–254, November 2006.

[131] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proc. IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, September 2001.

[132] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proc. 10th ACM Symposium*

on Architectural Support for Programming Languages and Operating Systems, pages 45–57, October 2002.

[133] B. Smith. ARM and Intel battle over the mobile chip's future. *IEEE Computer*, 41(5):16–19, May 2008.

[134] S. Somogyi, T.F. Wenisch, A. Ailamaki, and B. Falsafi. Spatio-temporal memory streaming. In *Proc. 36th IEEE/ACM International Symposium on Computer Architecture*, pages 69–80, June 2009.

[135] A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[136] Standard Performance Evaluation Corporation. SPEC CPU benchmark suite. http://www.specbench.org/osg/cpu2000/, 2000.

[137] Standard Performance Evaluation Corporation. SPEC OMP benchmark suite. http://www.specbench.org/hpg/omp2001/, 2001.

[138] Standard Performance Evaluation Corporation. SPEC CPU benchmark suite. http://www.specbench.org/osg/cpu2006/, 2006.

[139] P. Steenkiste. The impact of code density on instruction cache performance. In *Proc. 16th IEEE/ACM International Symposium on Computer Architecture*, pages 252–259, June 1989.

[140] J. Storer and T. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29:928–951, 1982.

[141] J. Suh and M. Dubois. Dynamic MIPS rate stabilization in out-of-order processors. In *Proc. 36th IEEE/ACM International Symposium on Computer Architecture*, pages 46–56, June 2009.

[142] Sun Microsystems. *The SPARC Architecture Manual Version 9*, 1994.

[143] P.K. Szwed, D. Marques, R.M. Buels, S.A. McKee, and M. Schulz. SimSnap: Fast-forwarding via native execution and application-level checkpointing. In *Proc. 8th IEEE Workshop on Interaction between Compilers and Computer Architectures*, February 2004.

[144] R.A. Uhlig and T.N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys*, 29(2):128–170, June 1997.

[145] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Considering all starting points for simultaenous multithreading simulation. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, March 2006.

[146] M. Van Biesbrouck, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, pages 45–56, March 2004.

[147] A. Varma, E. Debes, I. Kozintsev, P. Klein, and B. Jacob. Accurate and fast system-level power modeling. *ACM Transactions on Embedded Computing Systems*, 7(3), 2008.

[148] S. Vlaovic and E.S. Davidson. TAXI: Trace analysis for X86 interpretation. In *Proc. IEEE International Conference on Computer Design*, pages 508–514, September 2002.

[149] E. Wanderley Netto, R. Azevedo, P. Centoducatte, and G. Araujo. Multi-profile based code compression. In *Proc. 41st ACM/IEEE Design Automation Conference*, pages 244–249, June 2004.

[150] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. DRAMsim: A memory-system simulator. *Computer Architecture News*, 33(4):100–107, September 2005.

[151] V.M. Weaver. http://www.deater.net/weave/vmwprod/linux_logo/, 2009.

[152] V.M. Weaver and S.A. McKee. Are cycle accurate simulations a waste of time? In *Proc. 7th Workshop on Duplicating, Deconstructing, and Debunking*, pages 40–53, June 2008.

[153] V.M. Weaver and S.A. McKee. Can hardware performance counters be trusted? In *Proc. IEEE International Symposium on Workload Characterization*, pages 141–150, September 2008.

[154] V.M. Weaver and S.A. McKee. Can hardware performance counters be

trusted? Technical Report CSL-TR-2008-1051, Cornell University, August 2008.

[155] V.M. Weaver and S.A. McKee. Using dynamic binary instrumentation to generate multi-platform simpoints: Methodology and accuracy. In *Proc. 3rd International Conference on High Performance Embedded Architectures and Compilers*, pages 305–319, January 2008.

[156] I. Williams. An illustration of the benefits of the MIPS R12000 microprocessor and OCTANE system architecture. White Paper, SGI, 1999.

[157] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Proc.ACM International Conference on Measurement and Modeling of Computer Systems*, pages 68–79, May 1996.

[158] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In *Proc. IEEE/ACM 25th International Symposium on Microarchitecture*, pages 81–91, November 1992.

[159] C. Won, B. Lee, C. Yu, S. Moh, Y.-Y. Kim, and K. Park. Linux/SimOS - a simulation environment for evaluating high-speed communication systems. In *Proc. International Conference on Parallel Processing*, pages 193–199, 2002.

[160] Y. Wu, M. Breternitz, Jr., H. Hum, R. Peri, and J. Pickett. Enhanced code density of embedded CISC processors with echo technology. In *Proc. 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 160–165, October 2005.

[161] W.A. Wulf. Evaluation of the WM architecture. In *Proc. 19th IEEE/ACM International Symposium on Computer Architecture*, pages 382–390, 1992.

[162] Wm.A. Wulf and S.A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, March 1995.

[163] R.E. Wunderlich, T.F. Wenish, B. Falsafi, and J.C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proc. 30th IEEE/ACM International Symposium on Computer Architecture*, pages 84–95, June 2003.

[164] Xilinx. *MicroBlaze Processor Reference Guide*, 2004.

[165] M. Xu, R. Bodik, and M. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. In *Proc. 30th IEEE/ACM International Symposium on Computer Architecture*, pages 122–135, June 2003.

[166] X.H. Xu, C.T. Clarke, and S.R. Jones. High performance code compression architecture for the embedded ARM/THUMB processor. In *Proc. ACM Computing Frontiers Conference*, pages 451–456, April 2004.

[167] K.C. Yeager. The Mips R12000 superscalar microprocessor. White Paper, SGI, 2000.

[168] J.J. Yi, S. Kodakara, R. Sendag, D.J. Lilja, and D.M. Hawkins. Characterizing and comparing prevailing simulation techniques. In *Proc. 11th IEEE Symposium on High Performance Computer Architecture*, pages 266–277, February 2005.

[169] J.J. Yi and D.J. Lilja. Simulation of computer architectures: Simulators, benchmarks, methodologies, and recommendations. *IEEE Transactions of Computers*, 55(3):268–280, March 2006.

[170] J.J. Yi, R. Sendag, D.J. Lilja, and D.M. Hawkins. Speed and accuracy trade-offs in microarchitectural simulations. *IEEE Transactions of Computers*, 56(11):1549–1563, November 2007.

[171] M. Yourst. *PTLsim User's Guide and Reference*, 2007.

[172] M.T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, pages 23–34, April 2007.

[173] D. Zaparanuks, M. Jovic, and M. Hauswirth. Accuracy of performance counter measurements. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, pages 23–32, April 2009.

[174] H. Zeng, M. Yourst, K. Ghose, and D. Ponomarev. MPTLsim: a simulator for X86 multicore processors. In *Proc. 46th ACM/IEEE Design Automation Conference*, pages 226–231, 2009.

[175] Zilog. *Z80 family CPU User Manual*, 2001.

[176] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[177] A. Zmily and C. Kozyrakis. Simultaneously improving code size, performance, and energy in embedded processors. In *Proc. ACM/IEEE Design, Automation and Test in Europe Conference and Exposition*, pages 224–229, March 2006.