Accelerating Sampled Microarchitecture Simulation:

Rapid Warm Up for Simulated Hardware State

A Dissertation

Presented to

the Faculty of the School of Engineering and

Applied Science at the University of Virginia

in Partial Fulfillment for the Degree

Doctor of Philosophy

Computer Science

John W. Haskins, Jr.

Department of Computer Science

University of Virginia

Charlottesville, VA 22904

Approvals

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Computer Science

John W. Haskins, Jr.

Approved:

Kevin Skadron (Advisor)

John C. Knight (Chair)

John A. Stankovic

Marty A. Humphrey

Mircea R. Stan

Accepted by the School of Engineering and Applied Science:

Richard W. Miksad (Dean)

May 2003

Abstract

This dissertation introduces techniques for quantitatively reasoning about cache capacity and accelerating sampled microarchitecture simulation. By reducing the time spent warming up the simulated hardware state, Minimal Subset Evaluation warm up *(*MSEwarmup*) and its successor,* Memory Reference Reuse Latency *(MRRL) are able to substantially reduce overall simulation running times. Warm up is commonly used prior to modeling cycle-accurate simulation sample clusters to prevent* cold-start bias *from compromising the accuracy of simulated state in large structures like caches and branch predictor, and thereby preserve simulation accuracy. Unfortunately, warm up can be very time consuming, often representing 50% or more of total simulation time. Previous simulation strategies have warmed up the entire pre-cluster interval (*i.e., *modeled all cache and branch predictor interactions prior to each actual sample cluster) to obtain accurate hardware state; this is the* full-warmup *approach. While accurate, this time-consuming alternative may be prohibitive for large parameter-space searches. Other techniques have chosen a short but ad-hoc warm up length that reduces simulation time but may sacrifice accuracy.*

Minimal Subset Evaluation *(MSE) is a novel framework for quantitatively assessing the minimally sufficient number of unique memory references that must be handled within the cache in order to touch a certain proportion of the cache blocks with some user-chosen*

probability. (The aforementioned MSEwarmup *technique adapts MSE to forge a solution to the problem of accurately warming up L1 cache state.) This dissertation describes the mathematical underpinnings of MSE and demonstrates* MSEwarmup*'s use for quickly and accurately warming up both single-large-cluster and multiple-cluster simulation styles for small L1 caches. My experiments show that* MSEwarmup *yields errors of less than 1% in IPC measurements with cycle-accurate simulation.*

MRRL builds upon MSE and MSEwarmup*, but rather than using statistical methods, MRRL analyzes the lag time between consecutive references to each unique memory address in each pre-cluster–cluster benchmark partition. With this data MRRL is able to choose a point during the pre-cluster instructions to engage cache warm up (at all levels in the hierarchy, regardless of block size, associativity, or separation) and branch predictor warm up. Because of MRRL's applicability to all levels of the cache hierarchy regardless of organization as well as dynamic branch prediction, it supersedes* MSEwarmup *as a method for achieving accurate state quickly and accurately. MRRL yields an average error of less than 1% in IPC measurements relative to* fullwarmup *simulation, and reduces warm up by an average of 90% of the maximum potential speed up.*

Starting cache and branch predictor modeling late in the pre-cluster instruction stream allows both MSEwarmup *and MRRL to capitalize upon the observation that* only the branch predictor and cache interactions that occur nearest to a cluster are germane to simulation activity during the cluster itself.

Chapter 1

Introduction, Background and Motivation

This dissertation introduces Minimal Subset Evaluation (MSE) [17, 20] and Memory Reference Reuse Latency (MRRL) [18, 19]. MSE is a rigorous analytical framework for assessing cache occupancy based entirely upon the cache dimensions and the count of unique reference addresses handled within the cache. By calculating the number of unique references necessary to touch a certain proportion of L1 cache blocks, MSE can be used to accelerate sampled microarchitecture simulation. Rather than making a probabilistic determination of occupancy, MRRL attempts to measure the "amount of temporal locality." MRRL uses this information to accelerate sampled simulation by accelerating warm up in all levels of the cache hierarchy as well as the branch predictor.

Developing actual hardware is profoundly expensive. This practical consideration makes experimentation with new ideas in hardware prohibitive. Through simulation on the other hand, ideas may be developed, tested, discarded or refined repeatedly with infinitely more ease, speed, and at a modicum of the cost of such experimentation with hardware prototypes. It is precisely this vastly superior flexibility offered by simulation that makes software simulators a fundamental tool for computer architecture research. Simulation is especially beneficial for exploring radically new ideas that may not be feasible to prototype [43, 44].

Unfortunately, the flexibility of software simulation trades away speed. Detailed software simulation is orders of magnitude slower than native hardware execution. Thus, rather than simulating the full execution of a program in detail, researchers typically estimate the performance of a simulated microarchitecture by simulating only a sample of program *clusters* (contiguous segments of the dynamic instruction stream) in detail. By reducing the amount of time spent warming up simulated processor state, *MSEwarmup* and MRRL achieve further acceleration, thereby enhancing the value of software simulation as a tool for microarchitecture research.

This chapter introduces software simulation, strategies for making it more tractable, performance evaluation, declares my research thesis, and concludes with an overview of the research presented in this dissertation.

1.1 Software Simulation

Relative to native execution, simulation in software is very slow, introducing slowdown factors of hundreds or thousands even on today's fastest microprocessors. The amount of slowdown is inversely proportional to the level of detail captured by the simulation. Simple instruction-level simulation, for example, executes much more rapidly than highly detailed circuit-level simulation. This is because the ratio of native (*i.e.*, host) instructions to simulated instructions increases along with level of detail. In other words, simulation imposes a trade-off between speed and detail.

In software simulation, execution is emulated in a fetch–decode–execute loop: one by one, instructions are read from the binary, their operands acquired, and their prescribed operation simulated. However, whereas the elements of this loop are handled automatically in native hardware, many hundreds or thousands of native instructions are required to emulate these elements in software. In the simplest case, *functional* simulators such as SPIM [34] and the *sim-safe* component of the SimpleScalar [2, 4] software suite, model only the *architected state* of the microprocessor. For load–store architectures such as the MIPS [46] and the Alpha [52] (modeled by SPIM and *sim-safe*, respectively), this state consists of the contents of the register file and main memory. By tracking only the bare-minimum state necessary to execute a binary, functional simulation is one of the fastest software simulation methods. Functional simulators are excellent tools for compiler research, allowing developers to construct experimental compiler technology for non-native or unrealized hardware.

For more detailed simulation, it is necessary to maintain state beyond the register file and main memory. Experiments with cache or translation look-aside buffer (TLB) organization, for instance, require tracking cache and TLB state (*i.e.*, *sim-cache* [2]). Maintaining this state adds several hundreds more native instructions per simulated instruction beyond the aforementioned fetch–decode–execute loop for implementing lookup, detecting and rerouting misses, choosing a victim and performing evictions. Tracking branch predictor state (*i.e.*, *sim-bpred* [2]) introduces similar complexities and further augments the number of native instructions required per simulated instruction. An example of a simulator that models cache and branch predictor state in addition to architected state is the *Mipsy* component of the SimOS software suite [23].

Detailed studies that attempt to estimate the overall performance of an experimental microprocessor require *cycle-accurate* simulation. Cycle-accurate simulation models the step-by-step flow of instructions through a synchronous microprocessor pipeline. In synchronous microprocessors, instruction flow occurs discretely, synchronized by the pulse of an on-chip clock distribution network. Each pulse of the clock occurs at a constant rate (*e.g.*, 3.06GHz = 3,060 cycles/second) and is counted as one *cycle*. Hence for synchronous microprocessors, a key performance metric—instruction throughput—is measured in units of instructions per cycle (IPC) (*i.e.*, average number of committed instructions per on-chip clock pulse). (Other, less desirable metrics will be discussed in 1.3, and a case will be made for instruction throughput.) The research described in this dissertation studies the simulation of synchronous microprocessors.

Cycle-accurate simulation is terribly slow. With cycle-accurate simulations, running many of the SPEC95 benchmarks to completion with reference inputs takes days or weeks [54], and running some of the SPEC 2000 benchmarks takes many weeks even with today's fastest systems [29]. Because cycle-accurate simulation models the movement of individual instructions through the pipeline, the state of many more on-chip structures beyond the register file, main memory, cache, TLB and branch predictor must be maintained. In a wide-issue out-of-order pipeline, these structures include the instruction reorder buffer, issue logic, result forwarding network, register renaming hardware, functional units, store queue, and commit logic, to name just a few.

The apex of simulation detail is to model the flow of signals along wires to interconnected logic elements that compose on-chip structures and pipeline elements. For this level of detail, VHDL [45, 73] (Very High-speed Integrated Circuit Hardware Design Language) is the language of choice for coding the actual hardware description; and TyVIS [63] is an example of a software package that can simulate a microprocessor design coded in VHDL. Its cost is prohibitive for microarchitecture research; hence it is used for circuit-level work. The research discussed in this dissertation does not perform simulation in such detail; it is mentioned here for completeness.

To summarize, a microprocessor can be simulated at varying levels of detail, each more suitable to specific types of research. Functional simulation does not model any of the details of a specific processor organization beyond the architected state. This makes instruction-level simulators well-suited, for instance, to compiler research, enabling experimentation with non-native or even unrealized instruction sets. Microarchitectural simulators model the inner workings of all or part of an actual processor, including the pipeline, cache hierarchy, and branch predictor at the component level. These simulators are appropriate for estimating the performance of actual hardware components (individually and in collaboration), and are suitable for architecture and pipeline organization research. Finally, circuit-level simulators model the low-level implementation details of hardware components. Used in concert with the VHDL specification language, these simulators are useful for the development, analysis, and verification of architectural components.

1.2 Simulation Strategies: Sampling versus Reduced Inputs

To combat cycle-accurate simulation's long running times, most simulation strategies either take samples of multiple short *clusters* of contiguous groups of instructions from the dynamic instruction stream, "fast-forwarding" between clusters [8, 51], or else fast-forward to a single, large simulation cluster of 50–100 million instructions [50, 54]. Both techniques save time by executing in cycle-accurate detail only those instructions contained in the sample clusters. In contrast, during the pre-cluster phase (*i.e.*, during the emulation of instructions prior to each cluster) cycle-by-cycle modeling of individual instructions through the pipeline is not performed. The result is a much faster-running simulation.

Another approach to reduce simulation times is to use *reduced inputs* [20, 29, 30]: a shrunken input set based upon a benchmark's reference inputs, intended to execute in less time while exhibiting behavior similar to the full reference inputs. Reduced inputs however, raise the question of the reduced input's accuracy. In joint research with the KleinOsowski and Lilja at the University of Minnesota [20], we compare the accuracy of simulations with their MinneSPEC reduced inputs against sampled simulations using the original reference inputs for several SPEC CPU2000 benchmarks.

benchmark_input	IPC_{true}	$\text{IPC}_{sampling}$	$\text{IPC}_{reducedinput}$
art_110	0.5984	-0.45%	-9.09%
art_470	0.5974	0.64%	-8.94%
gzip_graphic	1.3645	-1.69%	3.79%
gzip_log	1.4620	-1.41%	7.91%
gzip_program	1.3884	0.44%	-1.52%
gzip_random	1.3187	1.39%	0.18%
gzip_source	1.3609	0.71%	1.06%
vortex_lendian1	1.0918	0.24%	-14.81%
vortex_lendian2	1.0573	-0.52%	-12.58%
vortex_lendian3	1.0890	-0.59%	-14.62%
vpr_place	0.8460	0.50%	11.42%
vpr_route	1.0232	1.73%	39.28%
MEAN		0.86%	10.43%

Table 1.1: Accuracy: Sampling versus reduced inputs [20]. Percent-error in IPC ($100\% \cdot \frac{(IPC-IPC_{true})}{IPC_{true}}$) measures the deviation from end-to-end cycle-accurate simulation (IPC_{true}). MEAN calculated from the absolute values of error measurements.

As Table 1.1 shows, sampling is more accurate, generating IPC values that are closer than reduced inputs to those obtained by complete, end-to-end cycle-accurate runs of the benchmarks on the reference inputs (IPC_{true}). The strategy of 50 equidistantly spaced samples of 10 million instructions apiece yielded a maximum error of 1.73% for *vpr_route* and an average error of 0.86%. The reduced inputs, however, did not consistently yield such high fidelity, with a maximum error of 39.28% for *vpr_route* and an average error of 10.43%. Table 1.2 shows that both reduced inputs and sampling significantly lessen simulation running time, with neither running for more than 4% of the time required for end-to-end cycle-accurate simulation. In terms of accuracy however, sampling is the clear victor; hence, my research focuses on this approach.

benchmark_input	t_{true}	$\%t_{sampling}$	$\%t_{reducedinput}$
art_110	756507 sec.	1.36%	3.06%
art_470	801815 sec.	1.40%	2.89%
gzip_graphic	829097 sec.	0.49%	1.73%
gzip_log	380475 sec.	1.24%	1.62%
gzip_program	953154 sec.	1.36%	1.76%
gzip_random	733661 sec.	1.17%	1.77%
gzip_source	673621 sec.	1.15%	1.95%
vortex_lendian1	890519 sec.	3.04%	1.49%
vortex_lendian2	971445 sec.	3.18%	1.37%
vortex_lendian3	954585 sec.	3.15%	1.39%
vpr_place	1050041 sec.	1.47%	1.53%
vpr_route	997469 sec.	1.18%	1.04%
MEAN		1.68%	1.80%

Table 1.2: Running time: Sampling versus reduced inputs [20]. Percentage of running time ($100\% \cdot \frac{t}{t_{true}}$) measures the fraction of end-to-end cycle-accurate simulation running time (t_{true}).

Table 1.3 documents further experiments, where I compared the running time of end-to-end cycle-accurate simulation to end-to-end cold (*i.e.*, modeling register file and main memory updates), and end-to-end warm (*i.e.*, modeling register file, main memory and cache interactions), in addition to the running times of two sampling disciplines: 50 equidistantly-spaced at 1 million instructions apiece and 10 equidistantly-spaced at 5 million instructions apiece. (These experiments were run on different, faster hardware than those from Tables 1.1 and 1.2; hence, the shorter running times for t_{true}.) End-to-end cold simulation is unwaveringly the fastest, in all cases executing in less than 11% the time required for end-to-end cycle-accurate simulation. End-to-end warm performs well also, completing in less than 20% the time required

benchmark_input	t_{true}	$\%t_{50\times1}$	$\%t_{10\times5}$	$\%t_{warm}$	$\%t_{cold}$
art_110	153937 sec.	12.01%	11.88%	15.84%	8.93%
art_470	169606 sec.	11.99%	11.89%	15.96%	8.93%
gzip_graphic	192591 sec.	12.35%	12.33%	18.10%	9.91%
gzip_log	72822 sec.	13.48%	13.78%	19.81%	10.70%
gzip_program	252959 sec.	13.10%	13.11%	19.07%	10.27%
gzip_random	152678 sec.	12.30%	12.29%	17.91%	9.76%
gzip_source	138742 sec.	13.14%	13.14%	19.15%	10.33%
vortex_lendian1	401396 sec.	13.13%	13.11%	19.21%	10.56%
vortex_lendian2	460819 sec.	13.10%	14.37%	19.09%	10.47%
vortex_lendian3	448293 sec.	13.60%	13.53%	19.19%	10.53%
vpr_place	274529 sec.	11.94%	11.85%	17.09%	9.51%
vpr_route	232330 sec.	11.17%	11.22%	16.84%	9.34%
MEAN		12.61%	12.71%	18.10%	9.94%

Table 1.3: Running time: 50×1 and 10×5 sampling versus end-to-end warm versus end-to-end cold. Percentage of running time ($100\% \cdot \frac{t}{t_{true}}$) measures the fraction of end-to-end cycle-accurate simulation running time (t_{true}).

by end-to-end cycle-accurate simulation. Notice however, that experiments from the two sampling disciplines complete in even less time than end-to-end warm simulation. These experiments implement *MSEwarmup*, one of the simulation acceleration techniques discussed later in this dissertation.

The accuracy of simulation within each cycle-accurate sample depends on avoiding *cold-start bias*. Cold-start bias is the name given the phenomenon whereby sampled data tend to be skewed when environmental state is inaccurate or unrepresentative at the beginning of a sample cluster [8]. In a microprocessor, performance is deeply affected by cache and branch predictor performance [21]: if the requested data are in the cache(s) and the branch predictor makes accurate predictions, then the processor

benchmark (suite)	fast-forward interval
bzip2 (SPEC CPU2000)	1,733
hydro (SPEC CPU95)	36
tomcat (SPEC CPU95)	144
vortex (SPEC CPU2000)	330
vpr (SPEC CPU2000)	746
wave (SPEC CPU95)	1,036

Table 1.4: Sample of pre-cluster intervals (in 100s of millions of instructions) used for simulations reported by Sherwood *et al.* [50].

will spend fewer clock cycles executing a program. Hence, to defeat cold-start bias the processor "environment" (*i.e.*, the state contained in the cache hierarchy and the branch predictor) needs to be as close as possible to the state that would have resulted from executing the entire pre-cluster phase in cycle-accurate detail. In other words, cold-start bias can be defeated by allowing a period of warm up prior to each cycle-accurate cluster.

One warm up technique for achieving accurate pre-cluster processor environment involves modeling all pre-cluster cache and branch predictor interactions, in addition to the architected state. I call this technique *fullwarmup* because it defeats cold-start bias by "warming up" the simulated cache hierarchy and branch predictor state throughout the pre-cluster instructions. The accuracy of the cache and branch predictor state under *fullwarmup* is unimpeachable; just as would have occurred using cycle-accurate simulation during the pre-cluster phase, all cache and branch predictor interactions are modeled. Thus, for the remainder of this research *fullwarmup* is used as the baseline reference for all accuracy measurements.

Unfortunately, while much faster than cycle-accurate simulation, *fullwarmup* is still expensive. As described before, modeling all cache and branch predictor interactions adds many more hundreds or thousands of native instructions beyond those required to model only the architected state. This causes the simulator to run longer and makes *fullwarmup* prohibitive, especially for large state-space searches requiring multiple simulations with varying parameters. *fullwarmup*'s expense is also a problem when the cycle-accurate clusters occur deep within a benchmark's dynamic instruction stream. To elucidate the latter point, consider Table 1.4 which shows the pre-cluster fast-forward distances prescribed by Sherwood *et al.* [50]. For each of these benchmarks, the target sample cluster begins many hundreds of millions of instructions from the start of execution.

1.3 Processor Performance Evaluation

Before describing the tools and techniques commonly used to evaluate microprocessor performance, it is essential to establish a metric by which performance can be usefully measured. As mentioned in 1.1, the metric of choice for this research is instruction throughput, measured in units of completed instructions per clock cycle. Two less desirable metrics [21, 60] are millions of instructions per second (MIPS) and billions of floating-point operations per second (GFLOPS).

The former is calculated as the quotient of dynamic instruction count divided by wall-clock running time (in microseconds) of the benchmark. While straight-forward to calculate, MIPS is not meaningful because dynamic instruction count can vary widely between different instruction set architectures. Say, for instance, that some program's source is compiled for two unique CPUs from vendor A and vendor B into binaries, bin_A and bin_B. If vendor A's CPU uses a minimalist instruction set that emphasizes simplicity whereas vendor B's CPU uses a richer instruction set, featuring operations that atomically perform complex manipulations, bin_A will likely have a larger footprint than bin_B. Finally, suppose that bin_A and bin_B on the same input, execute in the same amount of time; while neither vendors' CPU finishes sooner than the other, vendor A's CPU—using MIPS—would be branded the clear victor because it executed more instructions per unit of time on average, than vendor B's CPU. Although contrived, this example vividly illustrates the deceptive nature of the MIPS performance metric [21].

GFLOPS is a deceptive metric for similar reasons. Programs do not all share the same proportion of floating-point operations and therefore cannot be meaningfully compared. Software that calculates the stresses placed upon cables and anchor blocks in suspension bridges, for instance, will likely spend the overwhelming majority of its time performing numerous floating-point calculations; software that minimizes Boolean logic functions, however, will likely perform almost none. Hence, the bridge-modeling software will score much higher than the logic minimizer in GFLOPS units. Furthermore, back to the issue of instruction set architectures, the same source code compiled for two different CPUs may still yield two binaries with substantial differences in the count of floating-point operations if the two instruction sets do not implement the same floating-point operations. Those operations not implemented as native hardware instructions may be doable as a combination of simpler floating-point operations, emulated in software, or passed to an off-chip coprocessor. In either case, it is difficult to develop a compelling argument that the two binary's GFLOPS performance ratings are comparable.

For these reasons, MIPS and GFLOPS have been largely abandoned for scientific, scholarly discussion of microprocessor performance. In the context of software simulation however, still more complications arise from their use. Specifically, these metrics calculate a rate per unit time. While it is possible to develop a framework for mapping cycle-accurate simulation events to equivalent real-time durations, this process is lengthy and the results contingent on a plethora of unimportant technological peculiarities (*e.g.*, capacitance, feature size). Rather, software simulation requires a higher-level, more abstract notion of performance; instruction throughput meets this requirement nicely.

The average number of completed instructions per clock cycle measures performance by describing how fully a pipeline utilizes its maximum instruction retiring capacity. A superscalar pipeline design capable of retiring four instructions per cycle, for instance, has a theoretical maximum IPC of 4. By corollary, the minimum IPC of 0 indicates a CPU making no progress whatever. Instruction throughput is a very complete performance metric, well suited to software simulation. Particularly attractive is its independence from real-time considerations and that measurements of actual performance are tightly coupled to the theoretical maximum; the latter gives valuable context to all performance measurements, as the ratio of the measured IPC and the theoretical maximum IPC gives the pipeline's operational efficiency.

Furthermore, for a given clock speed, a benchmark's true running time is easily calculated by dividing the number of instructions executed by the product of instruction throughput and clock frequency, *i.e.*,

$$t = \frac{\#instructions}{(cycles/second)(IPC)}$$

The second part of processor performance evaluation is choosing the binary or set of binaries that will be executed to gather the performance measurements. Hennessy and Patterson [21] describe several categories of these *benchmark* binaries, *toy programs*, *kernels*, *synthetic programs*, and *real programs*. Toy programs are very small executables that implement simple algorithms such as sorting algorithms and binary search. Hennessy and Patterson immediately dismisses toy programs, calling them most appropriate as programming assignments for beginning programmers. Their tiny size and simplistic nature prevent them from posing any serious challenge worthy of performance measurements.

Kernels and synthetic programs are held in slightly higher regard than toy programs. Kernels are small, key components of actual programs. The Lawrence Livermore Loops [38] were once widely used for processor performance evaluation. Dhrystone [71] and Whetstone [10] on the other hand, are two examples of synthetic benchmarks. Synthetic benchmarks are artificial programs whose instruction mix is intended to match the instruction mix profile characteristics (*e.g.*, opcode frequency, basic-block size) of a large collection of real programs. Kernels are derived from actual programs, but may fail to exhibit the same characteristics of their parent program. Synthetic benchmarks are only very loosely related to any real software; thus, their performance appraisal cannot be expected to yield useful measurements that are representative of real programs.

By far, the most respected class of benchmark programs are real programs. The reasoning behind this is simple. Most users will not spend money on computer systems merely to run toy programs, or synthetic programs, or to extract the innermost loops of real software and only execute them. Most users will want to run real programs; therefore, the most reliable source of truly representative performance measurements will come from real programs. The range of real programs is quite large and very diverse, ranging from compute-bound software such as ray tracers to I/O-bound word processors. The general consensus among computer architects is to make performance judgments from measurements made on a wide assortment of real benchmark programs or *benchmark suites.* These benchmark suites have the advantage of masking the weaknesses of any one of its members by the inclusion of the others [21].

One very popular benchmark suite for evaluating CPU performance on compute-intensive (integer and floating-point) workloads is SPEC CPU [57, 58], from the Standard Performance Evaluation Corporation. The most recent edition, SPEC CPU2000, includes programs for compiling, combinatorial optimization, FPGA circuit layout and routing, chess playing, data compression, quantum chromodynamics modeling, shallow water simulation, image recognition, primality testing, computational chemistry, pollutant distribution, and several others. Earlier editions of the SPEC CPU benchmark suite were introduced in 1995, 1992, and inarguably in 1989 in response to the growing need for a realistic, uniform, performance evaluation standard.

As Hennessy and Patterson [21] describes, a particular set of benchmarks will likely not remain valid gauges of performance indefinitely. In their quest to address precisely this caveat of all benchmark suites, SPEC released these incarnations of SPEC CPU in an attempt to address several specific issues [9] including: running time, application size, and application type. The dynamic instruction count of several SPEC CPU95 benchmarks, while impressive at their introduction, were very tiny on more recent hardware, sometimes running for less than a minute. Such brief executions provided no challenge for contemporary microprocessor technology. In the five years since SPEC CPU 95, there had also been advances in software sophistication and complexity in addition to the advances made in hardware. This made it necessary to include programs with larger resource requirements as well as programs from new application areas.

As in earlier generations, SPEC CPU2000 programs are loosely grouped into two categories: integer and floating-point. Together, these two categories establish the aforementioned balance afforded by using a collection of multiple programs in the benchmark suite. The floating-point intensive programs keep balance by correcting the integer programs' inability to stress a pipeline's floating-point functional units. Secondly, since there are numerous special-purpose compiler optimizations geared toward floating-point computations, the integer programs maintain balance by preventing the performance evaluation from being unduly skewed in favor of these very

benchmark	t_{user}	t_{system}	%-system
art_110	246.700	0.188	0.076%
crafty	276.432	0.188	0.068%
equake	730.058	0.609	0.083%
facerec	482.358	0.693	0.143%
mgrid	665.514	1.871	0.280%
twolf	886.918	0.609	0.069%
vortex_lendian2	208.693	0.462	0.221%
vpr_route	407.248	0.393	0.096%

Table 1.5: Code execution time (in seconds) for several SPEC CPU2000 benchmarks executed on an Alpha 21264 running OSF/1. System code accounts for less than 0.3% of the overall execution time in all cases; therefore, system code is often ignored in software microprocessor simulation and performance evaluation.

specialized optimizations. Because of the good mix of real programs that stress floating-point and integer performance—both of which rely heavily on accurate branch prediction and good cache hit rates—I chose SPEC CPU95 and SPEC CPU2000 for my experiments.

The third part of microprocessor performance evaluation is operating system performance, but this piece of the performance puzzle is often ignored. While operating system code efficiency does impact the execution time of programs running on real hardware, for compute-intensive code this impact is usually insignificant. Table 1.5 shows the wall-clock running time of several SPEC CPU2000 benchmarks executed on an Alpha 21264 running OSF/1, for the user code (t_{user}), and the system code (t_{system}) as reported by the *time* UNIX utility, as well as the percentage of system code from the overall execution time (%-system).

The benchmark with the largest system time component is *mgrid*, but of the more than 11 minutes that this program executed, the 1.871 seconds of system time account for only 0.28% of the overall running time. This is the chief reason that the operating system contribution to the pipeline performance evaluation is largely ignored in microprocessor simulations; such a tiny contribution is safely ignored. (This is analogous to approximating the binomial distribution by the Poisson or Gaussian distribution.) It is difficult to justify the overhead of porting an operating system kernel, memory management infrastructure, and device drivers for such a small amount of additional precision. Even if operating system performance were significant however, meaningful comparative analysis would require that the operating system performance among various studies be nearly identical. Furthermore, inclusion of system-level performance would make the performance analysis more "impure" by making it subject to the influence of an additional source of noncomplicity outside the agreed upon suite of benchmark programs.

1.4 The Importance of Adequate Warm Up to Measurement Accuracy

The role of error and uncertainty in sampled microprocessor simulation is critical to establishing the significance of this research. Sampling produces error because only a subset of a population is measured rather than the entire population. As Conte *at el.* [8] show, however, *random cluster sampling* is a useful tool for microprocessor simulation, that is amenable to statistical methods which allow one to rigorously gauge the amount of error and to quantitatively express confidence in the result, based on the assumption that all members of the population had equal probability of being selected for inclusion in the sample. Specifically, random cluster sampling establishes a confidence interval [IPC$-\alpha$,IPC$+\alpha$] within which the true IPC can be assumed to exist with $X\%$ certainty (where α is a function of X).

For a well-chosen sample, a benchmark's true IPC—as would have resulted if the benchmark were simulated to completion in cycle-accurate detail—will fall inside the confidence interval computed by *fullwarmup* simulation. This is because modeling *all* cache and branch predictor interactions renders *fullwarmup* impervious to cold-start bias. As will be shown in Chapter 5, MRRL does well at mimicking the behavior of *fullwarmup*, deviating by a statistically insignificant[1] amount. On the other hand, for the exact same sample, an ineffective warm up technique can yield an estimated IPC that significantly deviates from the *fullwarmup* estimation, and whose confidence interval does not contain the benchmark's true IPC.

1.5 Research Preview

Checkpointing simulation state at the beginning of each sample cluster is one solution for accelerating sampled simulation while conquering cold-start bias; *fullwarmup* simulation would incur a one-time cost for each benchmark–input pair and the checkpoint would be loaded at simulator initialization. Unfortunately, separate checkpoints would be required for each desired combination of cache and branch predictor configurations, and for each set of simulation sample clusters for each benchmark program.

[1] I prove claims of statistically (in)significant deviation from *fullwarmup* by application of the matched-pairs t-test to actual benchmark simulations.

To avoid this unattractive alternative, my techniques make use of information about the branch-, instruction- and data reference streams. This will be discussed at length in Chapter 3 and Chapter 4.

While very reliable, modeling all pre-cluster cache and branch predictor interactions is unnecessary because of the principle of temporal locality [21, 27]. The chief observation, which makes both MSE and MRRL useful for warm up acceleration, is that *only the latest pre-cluster cache and branch predictor interactions will be relevant during the cluster itself.* Determining a contiguous subset of the pre-cluster instruction stream—bounded from above by the last instruction of the pre-cluster phase—whose branch predictor, data cache and instruction cache accesses are likely to impact the subsequent cluster allowed me to accelerate sampled simulation relative to *fullwarmup* by splitting the pre-cluster instruction stream into two distinct phases. In the first phase, the simulator only performs functional simulation, updating only the architected state for each instruction. In the second phase, the simulator additionally models cache and branch predictor interactions. Then, during the cluster itself, the simulator switches to cycle-accurate simulation. This is the three-phase simulation strategy used in numerous previous works [8, 17, 18, 19, 20, 28, 42]. The first, aggressively fast phase can be considered the "cold" phase; this is followed by the "warm" phase, where cache and branch predictor interactions are modeled; and concluded by the "hot" phase where cycle-accurate simulation of the processor pipeline takes place. This is illustrated in Figure 1.1.

Current pre-cluster acceleration approaches [8, 28, 42] use crude heuristics or cum-

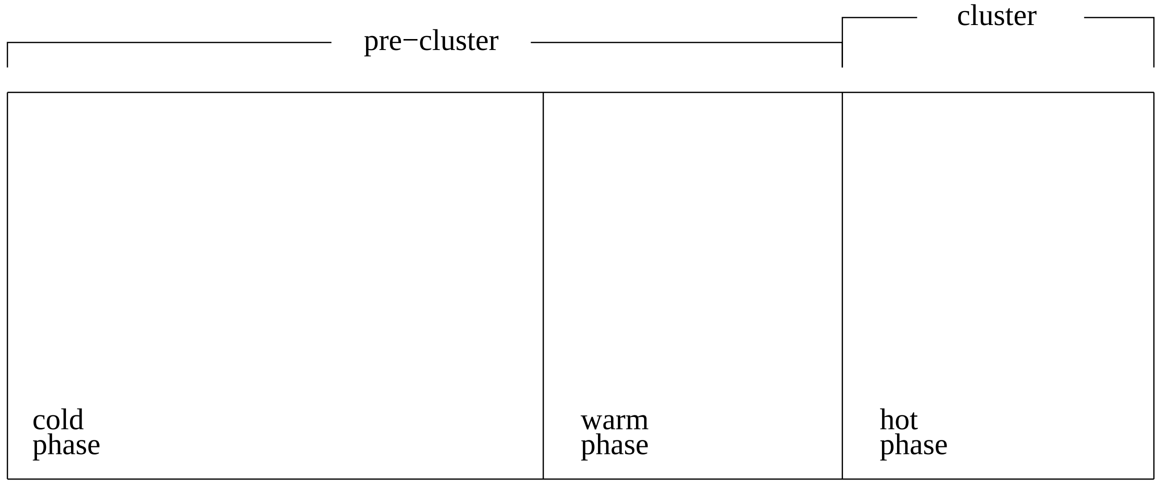

Figure 1.1: Pre-cluster–cluster pair subdivided into cold, warm, and hot phases. Cold phase models only architected state; warm phase models architected state, plus cache hierarchy, plus branch predictor; hot phase models pipeline in cycle-accurate detail.

bersome methods to decide the size (measured in completed instructions) of the warm phase. This motivates my

Thesis: Temporal locality suggests that those references occurring far from sample clusters are diminishingly useful for cache and branch predictor warm up. MSE can be adapted to mitigate cold-start bias by rigorously assessing a minimally sufficient number of unique memory references that must be handled within a cache in order to touch a proportion of cache blocks with probability $p \in (0, 1)$, thereby ensuring accurate warm up for large enough p. Similarly, MRRL ensures accurate cache hierarchy and branch predictor state by engaging warm up early enough to model the effect of references that fall within the reuse distance established by the measurement of temporal locality. The capacity of both to reduce overall running time by accelerating warm up while preserving accuracy will enable MSE and MRRL to render sampled microarchitecture simulation more efficient and therefore a more valuable tool for microarchitecture research.

The rest of this dissertation is organized as follows. The next chapter presents related work. Chapter 3 presents MSE; Chapter 4 presents the MRRL approach. Chapter 5 presents the experimental methodology and results. A detailed description of the software tools that were developed in the course of this research are described in Chapter 6. Finally, research contributions are summarized, the dissertation concludes, and potential avenues for future work are discussed in Chapter 7.

Chapter 2

Related Work

2.1 Benchmark Sampling Strategy

Because simulating benchmarks end-to-end in cycle-accurate detail is prohibitive, several studies have explored ways to simulate only portions of the program's overall execution in cycle-accurate detail. Skadron *et al.* [54] used a sequence of heuristics to find a single, short but representative simulation window of 50 million instructions. The most important component of their approach is to exclude unrepresentative start-up behavior from early in the benchmark's execution; Skadron *et al.* [54] go on to present a table of fast-forward instruction counts for the SPECInt95 benchmarks.

Lafage and Seznec [31] modify the sampling approach by using statistical classification methods to characterize the entire benchmark and provide a more rigorous guarantee of the chosen sample's representativeness. A potential problem with this approach is that finding configuration-independent metrics for representativeness is difficult.

Sherwood *et al.* [50] propose Basic Block Distribution Analysis (BBDA). Their technique profiles the execution frequency characteristics of a benchmark's basic blocks in order to isolate a continuous subset of the dynamic instruction stream whose execution characteristics closely mimic the complete, end-to-end cycle-accurate execution of the benchmark. BBDA's key insight is that periodic basic block execution frequency reflects the periodicity of various architectural metrics such as IPC, cache miss rate, and branch predictor accuracy in cycle-accurate simulation. In subsequent work, Sherwood *et al.* [51] build upon the BBDA concept to create a technique that automatically isolates multiple contiguous subsets of the dynamic instruction stream since some benchmarks' behavior is too complex to be characterized by a single instruction stream slice. In both cases, their aim is to reduce simulation running times by only executing in cycle-accurate detail, a small representative subset of the dynamic instruction stream.

Conte *et al.* [8] take a different approach, and instead simulate multiple fixed-length clusters of instructions selected randomly from the dynamic instruction stream. Because the execution clusters are chosen randomly (*i.e.*, such that all parts of the dynamic instruction stream have equal probability of being chosen), *random cluster sampling* is amenable to statistical analysis and allows the determination of a confidence interval. If one can safely assume that cold-start bias does not adversely impact simulation accuracy, then with $X\%$ confidence, the true benchmark IPC is within the confidence interval.

2.2 Solutions to Cold-start Bias

Key to Conte *et al.*'s [8] technique is ensuring the accuracy of the state in large structures like the caches and branch predictor. Their work addresses branch prediction structures (assuming a perfect cache) and suggests that recycling stale predictor state[1] from the previous cluster plus a short warm up interval [32] of at least 7,000 instructions prior to the next sample cluster is sufficient to minimize cold-start bias in the branch predictor and achieve very small errors of a few percent between the mean estimated IPC and the true IPC. Conte's warm up approach is referred to as *shortwarmup* in Chapter 5. Incidentally, because Conte does not treat cache effects, assuming a perfect cache in the experiments presented, MSE and MRRL address exactly the problem that Conte's method does not.

Co and Skadron [7] revisit the problem of warming up the branch predictor, but in a more modern scenario, addressing context switching. They investigated whether a process's performance could be hindered by branch predictor state loss resulting from a second (or more) process's acquisition of the CPU when the initial process reclaims the CPU and is allowed to resume. Their data show that due to the high clock speeds of modern CPUs, enough instructions are executed per quantum to reestablish branch predictor state, rendering the adverse effects of context switching induced state loss negligible. Specifically, they show that even large branch predictors train in as little as 128K instructions—a mere fraction of the instructions executed per quantum in a

[1] By "stale state," I mean that the hardware state as it appeared at the conclusion of the prior cluster is used as the starting state of the current cluster.

modern, fast CPU—even when a process's branch predictor state is completely lost prior to reclaiming the CPU.

As a part of their PARSIM parallel microprocessor simulation system, Nguyen *et al.* [42] develop an analytical technique for computing warm up length. Their formula calculates a function of the cache block width, associativity, the average population density of memory references throughout the instruction stream, and the average steady-state cache miss ratio, the latter of which unfortunately, implies a one-time *fullwarmup* run to measure it. MSE improves upon this by making calculations based only upon the cache dimensions (*i.e.*, the number of sets, and degrees of associativity) and is therefore free from even a one-time *fullwarmup* cost. PARSIM is also a trace-driven system. In response to the much increased speed of microprocessors, however, the Standard Performance Evaluation Corporation (SPEC) massively increased the running length of the benchmarks chosen for the SPEC CPU2000 suite [9]. Generating, storing, and accessing the resultant enormous traces is unattractive in terms of storage and access cost. MSE and MRRL were purposefully designed to avoid this expensive requirement.

Other heuristics for reducing cold-start bias are studied by Kessler *et al.* [28]. They consider using half of the pre-cluster references for warm up purposes; tracking only entries that are known to contain good state; using stale state from the previous cluster; and flushing state but estimating how much error this introduces.

2.3 Analytical Simulation Frameworks

Thiébaut [66] describes premier work on the analytical assessment of the memory reference stream and draws an insightful analogy between memory access patterns and fractal random walks on the one-dimensional lattice. In Thiébaut's model, the one-dimensional lattice is emulated by main memory, and the stride of next reference is described by random variable U; the "walk" of the memory reference stream is fractal if [37]:

$$P[U > u] = \left(\frac{u}{u_0}\right)^{-\theta}, u \geq u_0$$

where u_0 and θ are constants computed from an analysis of a program trace. θ is called the *fractal dimension* of the random walk and describes whether the random walk tends to sparsely ($\theta < 1$) or repetitively ($\theta > 1$) visit cells of the one-dimensional lattice. From this framework, Thiébaut describes a method for accurately predicting the miss ratio of fully associative caches. In later work [68], Thiébaut *et al.* builds upon the fractal random walk framework to address the generation of purely synthetic memory reference traces which accurately mimic the miss ratio of real program traces. Finally Mendelson in collaboration with Thiébaut and Pradhan [39], describe an analytical model for predicting the proportion of live[2] cache lines. Once again, the authors build upon the fractal random walk using the hyperbolic probability distribution function given above to develop a model that estimates the cache's steady state

[2]By *live* I mean a cache line that will be accessed at least once before being invalidated, evicted, and refilled.

behavior. All three of these works describe techniques that are dependent upon the simulated cache block width. MSE shares this dependence; MRRL improves upon this by being entirely independent of block width.

Strecker [62] extends the work of Easton *et al.* [11], who present a formal model for computing the cold-start miss ratio of a fully-associative cache from the steady state miss ratio. Strecker argues however, that in a multitasking environment a tumultuous relationship exists between cache blocks belonging to different processes. If it can be safely assumed that 100% of all cache blocks are valid, this results in numerous capacity misses when an interrupted process resumes execution. (Thiébaut [67], calls the collection of cache misses that occur when a process reacquires control of the CPU the *cache-reload transient.*) Strecker's model presents an analytical framework for estimating the miss ratios of direct-mapped caches for multiple programs in a multitasking environment based on each program's instantaneous miss ratio. Thiébaut *et al.* present the *reload-transient model* [67]. The reload-transient model probabilistically estimates the cache-reload transient of two processes in a multitasking environment as a function of the footprints[3] of the two competing processes and the dimensions of the cache (*e.g.*, number of sets, degrees of associativity). MSE shares a dependence on the dimensions of the cache; MRRL is independent of the cache dimensions.

Wood *et al* [72] establish the concept of *cache generations.* Each cache generation begins immediately after a new line is brought into the cache and ends when the line

[3]By *footprint*, I refer to the number of cache lines belonging to a process.

is evicted and replaced. Their notion of cache generations establishes a framework for analytically estimating the *unknown* or *cold-start* reference miss ratio, μ. They further establish that μ is substantially higher than the miss ratio of references chosen at random. Armed with reliable $\hat{\mu}$—estimated unknown reference miss ratio—they were able to accurately measure cache miss ratios in sampled trace-driven simulations. Rather than attempting to estimate cache miss ratios, MSE and MRRL more directly address the specific issue of determining how much warm up is necessary to preserve simulation accuracy. MRRL is furthermore applicable to branch predictor warm up.

2.4 Important Insights from Cache Design

In their Cache Decay research, Kaxiras *et al.* [27] propose a technique of cutting power to (heuristically presumed) dead cache lines, thereby reducing leakage power. Their measurements show that for a 32KB L1 data-cache, the proportion of the cache lines' dead time ranges from 45% to as much as 99% for the SPEC CPU2000 benchmarks. Their work shows that most cache lines' active lifetime is significantly longer than their useful lifetime.

Lai *et al.* [33] describe a new hardware mechanism called the *dead block predictor* (DBP) which—as its name suggests—heuristically estimates a cache block's final reference prior to being invalidated, evicted, and refilled. The DBP's operation is somewhat analogous to dynamic branch prediction. Each cache block is paired with its own *dead block signature*: an encoded trace of the memory reference stream since the cache block was first fetched. A table of two-bit saturating counters is used to gauge confidence in the dead block predictions. Their measurements show that predicting the useful lifespan of individual cache blocks and initiating new block prefetches well in advance of simple demand-fetching, results in substantial CPU performance improvements.

2.5 Related Work Synopsis

In short, several techniques exist for sampling execution ([8, 31, 50, 51, 54]); these methods demonstrate the effectiveness of sampling in reducing simulation times while preserving accuracy, relative to end-to-end cycle-accurate simulation. Nevertheless, all these techniques are dependent on accurate *warm up* of the cache and branch predictor state prior to each sample. While some heuristics for determining the amount of the pre-cluster instruction stream to warm up have been described ([7, 8, 28, 42]), I am not aware of any efforts to develop a more formal approach to minimizing warm up lengths while preserving accuracy besides the cumbersome trace-driven technique described by Nguyen [42]. Analytical frameworks for reasoning about cache behavior have also been studied ([11, 39, 62, 66, 68, 72]), but these approaches only offer insight into steady-state behavior. Finally, recent developments ([27, 33]) in cache design propose insightful clues for determining the useful lifespan of individual cache blocks.

Chapter 3

Minimal Subset Evaluation

Minimal Subset Evaluation [17, 20] (MSE) seeks to quantitatively assess the probability of touching a certain fraction of cache blocks based only on the cache dimensions and the count of unique memory reference addresses. This capability can be exploited to determine warm phase length necessary to warm up a small L1 cache. Many fewer native instructions are executed per simulated instruction in the cold phase than in the warm phase (both of which combine to form pre-cluster phase); hence, accurate speedup is achieved by expanding the length (in instructions) of the cold phase while leaving enough of the warm phase to accurately establish L1 cache state, allowing the cycle-accurate hot phase to execute, confident that cold-start bias has been defeated.

3.1 MSE Warm Up

The *MSEwarmup* technique is an adaptation of Minimal Subset Evaluation to the problem of deciding warm phase length. *MSEwarmup* makes this determination from probability computations and data about the memory reference streams for instructions and data. The *MSE formula* determines the probability that handling m cache accesses by unique reference addresses will touch a given proportion of cache blocks. The steps of the adaptation for *MSEwarmup* is enumerated below:

1. First, the user chooses the location of simulation sample clusters within the benchmark. (Some approaches choose just a single cluster [31, 50, 54] whereas others choose several [8, 17, 18, 19, 42, 51].)

2. The user selects a desired probability of accuracy $p \in (0, 1)$. This is a goal value. Iteratively, calculations are performed to determine the minimal m necessary to achieve probability p of touching a given proportion of the cache blocks.

3. The user profiles the benchmarks to characterize the occurrence of unique[1] memory references. This is a one-time cost for each set of sample clusters from a benchmark–input pair; these profiles are valid for any p, and L1 cache configuration.

4. The user then examines the profile to determine, for each simulation sample cluster, how many total instructions, t, prior to the sample must be executed in order to observe the aforementioned MSE-prescribed m unique references.

5. The simulation can then be run in aggressive cold mode consisting of only functional simulation in which just architected state is simulated. At t instructions prior to the beginning of the cycle-accurate hot phase, the cold mode changes into warm mode, in which interactions with the L1 cache are also modeled.

[1] By “unique” I mean that no two memory references access the same address.

Then, once the cluster is reached, the cache structures will with probability p have the desired proportion of blocks touched.

To thoroughly discuss MSE it is helpful to define several more variables. Let N be the number of sets in the cache and a be its associativity. (For direct-mapped caches, $a = 1$.) The MSE formula is then used to determine m for any p, a and N; that is, $m = \text{MSE}(N, a, p)$.

Once a probability of accuracy p has been selected, the MSE formula is calculated to arrived at the MSE-prescribed number of uniques (m). From this, t—the number of instructions to execute during the warm up interval that contains m uniques—is determined. t could be chosen, for instance, from a trace of the memory reference stream. Such traces, however, rapidly become large and unwieldy. Elnozahy [12] has addressed the cumbersome nature of traces and offers approaches that make their use more viable. Instead of dealing with full (or even compressed) traces however, I obtain t using data gathered from the *MSEprofiles*.

As enumerated above, *MSEwarmup* uses a two-pass process. First, I run software that I have developed which profiles the occurrence of unique memory references in the instruction- and data reference streams of each pre-cluster–cluster pair, packing that data into MSEprofiles. Second, when the cold–warm–hot simulator itself is launched, it reads the MSEprofile for the appropriate benchmark–input pair and begins simulation. The profiler works by maintaining an associative array for the stream of instruction memory references and for the stream of data memory references. Each associative array element operates like a single cache block of a fully-associative cache whose cache block size is equal to the block size used by the cold–warm–hot simulator[2]. Each time a memory reference occurs, the corresponding associative array entry is logically time-stamped with the completed instruction count. At the conclusion of the profiling run, the set of timestamps are sorted in descending order; the timestamp occurring m-th in the list is the number of instructions (t) prior to the cycle-accurate sample that must be executed in order to encounter m unique memory references. In other words: $t = \mathit{MSEwarmup}(N, a, p, \text{MSEprofile})$.

In the Chapter 5 experiments, I make the assumption that the pre-cluster phase is long enough and accesses enough unique reference addresses that all Na cache blocks are touched at least once prior to the cluster. Hence, MSE is used to find the warm up interval t that touches all Na cache blocks. In general, this is not a good assumption. Some benchmarks will not touch all entries in a cache, especially a large cache, regardless of the length of the pre-cluster interval; this is what limits MSE to warming up only smaller primary (L1) caches. The problem is that for some fixed p, larger caches require larger m to touch the designated proportion of cache blocks. Thus, it becomes increasingly less likely that m or more unique references are accessed during the pre-cluster phase. If fewer than the MSE-prescribed m unique references exist among the pre-cluster instructions, a safe, conservative fallback is to let $t = 0$. That is, when fewer than m uniques are available for the current pre-cluster–cluster

[2] A common block width among contemporary L1 instruction- and data caches is 32 bytes; this is the block size used for the experiments. Dependence on the cache block width is actually a limitation of *MSEwarmup* that is remedied by MRRL.

pair, *MSEwarmup* reverts to *fullwarmup*. This dearth of unique references is common when attempting to warm up secondary (L2) caches which tend to be much larger than L1s. To accommodate this, MSE employs two more variables, α, $\beta \in (0, 1]$. These parameters tune N and a in the following way: $t = MSEwarmup(\alpha N, \beta a, p,$ MSEprofile). Thus, if the user needs to calculate the number of unique references necessary to touch only a fraction of sets and a fraction of blocks within each set, $\alpha < 1$ and $\beta < 1$.

The next section contains a thorough discussion of several important assumptions and the critically important uniformity assumption that implicitly underlies the MSE formula; the MSE formula itself, is discussed thereafter.

3.2 MSE Assumptions

The MSE formula ($m = \text{MSE}(\alpha N, \beta a, p)$) calculates the probability that at least αN sets of a cache will be touched at least βa times. The formula however, is based on the assumption that *unique memory references are typically distributed uniformly throughout the cache*. This assumption does not contradict the well-known, empirically demonstrated principle of locality. The critical difference is that the locality principle considers the entire stream of memory references, L. My assumption, by contrast, refers only to the subset of the memory reference stream that does not contain duplicates, $unique(L)$.

Indeed, uniform distribution of unique memory reference addresses is exactly the ideal behavior for a cache because this would reduce the likelihood of conflicts (*i.e.*, having two different references map to the same location within the cache). In an ideal direct-mapped cache, for instance, if all cache blocks are valid, data in any set would have a $\frac{N-1}{N}$ chance of surviving a unique incoming reference address; the larger the N, the greater the chance of survival.

To verify the uniform distribution of $unique(L)$ throughout the cache analytically, I employed the χ^2 goodness-of-fit test [13, 59, 64]. I developed software that counts (based on some assumed cache block width) the number of references to each cache set among the unique references only. From this, I first tallied the total number of unique memory accesses, $|unique(L)|$, and calculated a best estimate expected number of accesses per set

$$\bar{x} = \frac{|unique(L)|}{N}$$

Using $\bar{x}$, sets are grouped into bins such that the best estimate average number of accesses per bin is at least 5 [13, 64]. Finally, I used these data to compute χ^2_o: the observed χ^2. For all benchmarks and for every cache configuration tested, the raw profile data passes the χ^2 test of the null hypothesis that the distribution of cache accesses among the unique references is uniform at the 1% level of significance. Table 3.1 gives the χ^2 values computed for each benchmark, each cache configuration ($N \in \{256,512,1024,2048\}$), the degrees of freedom and the corresponding 1% critical value.

I also performed the same test on the profile data from the SPEC CPU2000 benchmarks. Experiments on these benchmarks use multiple-cluster simulation; thus I was

$N = 256$; $df = 254$, $cv_{0.01} = 309$		
	χ^2_o	$P(\chi^2_{0.01,254} \geq \chi^2_o)$
compress	0.254	1.000
gcc	149.098	1.000
go	296.164	0.036
ijpeg	135.636	1.000
m88ksim	3.556	1.000
perl	41.148	1.000
$N = 512$; $df = 510$, $cv_{0.01} = 587$		
	χ^2_o	$P(\chi^2_{0.01,510} \geq \chi^2_o)$
compress	0.000	1.000
gcc	140.250	1.000
go	0.000	1.000
ijpeg	161.670	1.000
m88ksim	11.220	1.000
perl	149.940	1.000
$N = 1024$; $df = 1022$, $cv_{0.01} = 1130$		
	χ^2_o	$P(\chi^2_{0.01,1022} \geq \chi^2_o)$
compress	3.066	1.000
gcc	945.350	0.958
go	0.000	1.000
ijpeg	247.324	1.000
m88ksim	0.000	1.000
perl	357.700	1.000
$N = 2048$; $df = 2046$, $cv_{0.01} = 2197$		
	χ^2_o	$P(\chi^2_{0.01,2046} \geq \chi^2_o)$
compress	0.000	1.000
gcc	75.702	1.000
go	0.000	1.000
ijpeg	619.938	1.000
m88ksim	0.000	1.000
perl	716.100	1.000

Table 3.1: $\tilde{\chi}^2_0$ for SPECInt95 benchmarks and various N; df is the number of degrees-of-freedom; $cv_{0.01}$ is the critical value of χ^2_o at the 1% level of significance.

forced to perform the χ^2 test for each pre-cluster phase individually. The results were similar to those presented in Table 3.1: statistically close to uniform for the MSE formula to yield reliable results. This assessment is additionally supported by the successful application of *MSEwarmup* to multiple-cluster simulations (see Table 5.6).

A possible hole in the uniformity assumption is second-level (L2) caches. L2 caches typically have a much larger volume and greater cache block capacity than L1 caches. L2s are furthermore, typically *unified*, hosting instructions and data. It is possible that some programs' instruction- and data reference streams will destructively interfere with each other in a unified L2, invalidating the assumed uniform distribution of unique memory references. Hence, a second assumption is that the MSE formula is performing calculations based on a non-unified cache.

The MSE profiler assumes a specific cache block width. For the *MSEwarmup* experiments I chose a block size of 32 bytes. (This has become common in the L1 instruction- and data caches of some contemporary microprocessors including the Pentium III [47], and UltraSPARC III [25]. Hence, for my experiments, L1 caches are configured to use 32-byte blocks.) Because of the nature of cache behavior—manipulating not single words or bytes of data, but whole data blocks—block width defines uniqueness among memory references. Block width uniqueness for a 32-byte block, byte-addressable cache, for example, is distinguished by the high 27 address bits. (In other words, uniqueness is determined on the basis of the remaining address bits after discarding the $\log_2 32 = 5$ least significant bits.) Restricting the profiler to this single assumption was a conscious design decision that maximizes

MSE's applicability; the only other cache dimensions—number of sets and degrees of associativity—are perfectly flexible. Hence, the m returned by the MSE formulas for a given p refers to a number of unique references based on block width granularity.

One final assumption, made not by MSE, but the MSE adaptation to cache warm up is a method for discovering what proportion of sets ($\alpha \in (0, 1]$) and what proportion of blocks within each set ($\beta \in (0, 1]$) would have been touched by *fullwarmup a priori*. Unfortunately, block size alone is insufficient to discover α or β during the profiling run.

Mendelson, Thiébaut, and Pradhan [39] developed a technique for analytically predicting the steady-state proportion of live blocks within a cache. Steady state behavior, however, may yield inaccurate results for *MSEwarmup* since the end-to-end execution is partitioned into individual pre-cluster–cluster pairs. What if, for instance, the live–dead behavior of the cache on a particular pre-cluster–cluster pair diverges significantly from the steady state? Furthermore, Mendelson *et al.* [39] do not determine liveness and deadness with respect to the number of sets and number of blocks per set that are live, just the percentage of all cache blocks that are live. Additionally, MSE's definition of liveness differs subtly from Mendelson's. MSE's notion of a live cache block is not merely one that will be reaccessed before being evicted, but one that will be reaccessed *during the next cluster* before being evicted. For these reasons, the techniques described by Mendelson *et al.* [39] are not obviously useful for helping *MSEwarmup* discover α and β.

If unable to find α and β *a priori*, one is forced to either guess, but unsubstantiated guessing may yield inaccurate results; or suffer a one-time cost of executing *fullwarmup* to discover α and β, but this defeats the purpose of finding a technique that makes *fullwarmup* unnecessary; or set $\alpha = \beta = 1$. While the latter choice is very conservative, forcing some pre-cluster–cluster pairs to revert to *fullwarmup*, it does not sacrifice accuracy and, for small enough caches—still simulates in less time than *fullwarmup*.

3.3 The MSE Formula

The MSE formula calculates the probability that $\alpha N \beta a$ blocks of an a-way associative cache with N sets will be touched at least once, thus:

$$p = \frac{\sum \left[\binom{m}{x_1, x_2, \ldots, x_{N-1}} \middle| \, s.t.\ at\ least\ \lceil \alpha N \rceil\ x_j \geq \lceil \beta a \rceil \right]}{\sum \binom{m}{x_1, x_2, \ldots, x_{N-1}}}$$

The denominator in the formula is a count of the number of ways to touch each of the N sets in varying combinations, such that the sum of touches is equal to m. The numerator is very similar, but filters away all cases for which more than αN sets have fewer than βa blocks per set are touched. Their quotient is the probability of touching at least βa blocks of at least αN sets, once or more.

The formula gives insight into the importance of m representing a number of *unique* reference addresses. The MSE formula makes a probability calculation of the number of sets that get touched and how many times each set is touched; duplicate references are irrelevant because duplicates will always map to exactly the same set in the cache.

Notice the absence of absolute bounds on sum in both the numerator and the denominator. This is intentional and implies that the sum is over all valid configurations of lower terms in the multinomial coefficient. (Basically, the sum of the lower terms is less than or equal to m to be valid.) The complexity of this computation is bounded by the complexity of calculating the sum in the denominator. The problem with performing this computation however, is precisely the absence of explicit bounds on the denominator sum which forces one to account for all valid combinations of lower terms in the multinomial coefficient.

To combat this, I have tested brute-force multithreading techniques in conjunction with optimizations that exploit combinatorics to avoid "double-counting," *e.g.*, $\binom{8}{1,2,4} = \binom{8}{2,4,1}$. All these were ineffective however, as the number of valid multinomial configurations goes exponentially in m. For instance, when executed on a dual-CPU, 500 MHz Pentium III system this calculation took more than three weeks for $m = 64$! If even this trivial number of unique references is intractable to work with, this formula is useless in general. Hence, I developed the *direct-mapped MSE approximation* (so called because it assumes $a = 1$):

$$p = 1 - \frac{\sum_{k=1}^{\lceil \alpha N \rceil - 1} \binom{N}{k} k^m}{\sum_{k=1}^{\lceil \alpha N \rceil} \binom{N}{k} k^m}$$

The numerator in this formula counts the number of ways to touch only k or fewer sets for $k \in [1, \alpha N - 1]$. In other words, the numerator counts the number of ways to *fail* to touch αN sets. The denominator on the other hand, counts the number of ways to touch as many as αN sets (*i.e.*, $k \in [1, \alpha N]$). In other words, the denominator counts the number of ways to fail plus the number of ways to succeed to touch αN

sets. Their quotient is the probability of failing to touch at least αN sets at least once; 1 minus this quotient is the probability of success.

I developed software that can calculate the direct-mapped approximation relatively quickly even on an older, 180MHz Pentium Pro—the longest calculation I tried took approximately 30 minutes. The formal MSE formula can be easily estimated by multiplying the result of the direct-mapped MSE approximation by the associativity, *i.e.*, $m = a \cdot$ MSEapprox$(\alpha N, 1, p)$. This approximation essentially emulates the case where the first m uniques successfully touch the required αN sets; the second m uniques touch αN sets; ...; the a-th m uniques touch αN sets. This approximation's imprecision is due to the fact that it does not ensure that the αN sets touched after each bundle of m uniques is the same αN sets touched by the previous m uniques. However, in the case that $\alpha = 1$, this is a good and indeed conservative approximation.

Unfortunately, I have not found a closed-form solution for m given p for either the MSE formula or the direct-mapped approximation. Instead, the software that I have written iteratively tests values of m to find the appropriate m for a specified p, αN, and βa.

3.4 Minimal Subset Evaluation Summary

Minimal Subset Evaluation (MSE) is a tool for analytically reasoning about cache occupancy based on the dimensions of a cache and the number of unique memory references handled within the cache. Based on the assumption that unique references are dispersed uniformly throughout the cache (as demonstrated in 3.2), the MSE formula calculates the probability of touching αN cache sets and βa blocks within each set, where N is the number of cache sets, a is the set associativity, and α, $\beta \in (0, 1]$. In other words, the MSE formula is useful for computing the probability that m unique references will touch a certain proportion of cache sets, and a certain proportion of blocks within each set. Unfortunately, the computational complexity of the MSE formula is intractable, but the MSE direct-mapped approximation formula makes the simplifying assumption that each set holds a single cache block, and can be computed quickly. A good estimate of the number of unique references necessary to touch $\alpha N a$ blocks with probability p is am, where m is the number of unique references necessary to touch αN sets of a direct-mapped cache as computed by the direct-mapped approximation.

MSEwarmup adapts MSE for accelerated warm up by determining the number of unique references necessary to touch a $\alpha N \beta a$ cache blocks with probability p. By profiling the count of unique references among the pre-cluster instructions, *MSEwarmup* determines the warm phase duration of t instructions that contains m unique memory references. Warming up the m unique references immediately preceding a sample cluster, cache state will accurately establish cache state with probability p.

Chapter 4

Memory Reference Reuse Latency

Memory reference reuse latency [18, 19] (MRRL) is a technique for accelerating sampled microarchitecture simulation which builds upon insights gained during the development of Minimal Subset Evaluation. MSE was developed to quantitatively assess cache occupancy as a function of the cache dimensions and the count of unique references handled within the cache. The adaptation of this capability to establishing good cache state (the *MSEwarmup* technique) unfortunately, is inconvenient. MSE's uniformity assumption is only verified for small L1 caches (see Chapter 3), which limits the scope of *MSEwarmup*. The requirement that profiling occur with respect to a specific block-width granularity can also be awkward. Even though 32-byte blocks are standard fare for contemporary microprocessors' L1 instruction- and data caches [25, 47], some microprocessors have different L1 cache block sizes [40]. To apply *MSEwarmup* it would be necessary to profile the benchmark twice according to two block widths or, just as clumsy, profile according to both block granularities

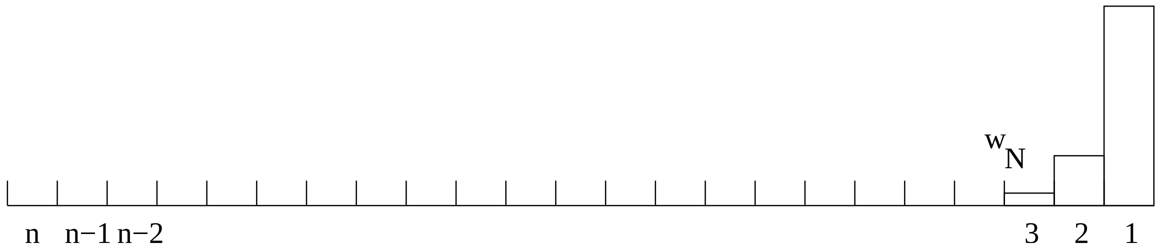

Figure 4.1: Reuse latency histogram of n mutually exclusive partition buckets of the discrete interval $[1, L]$ (where L is the number of instructions in the pre-cluster–cluster pair); $N \times 100\%$ of all references have reuse latencies of w_N or fewer instructions.

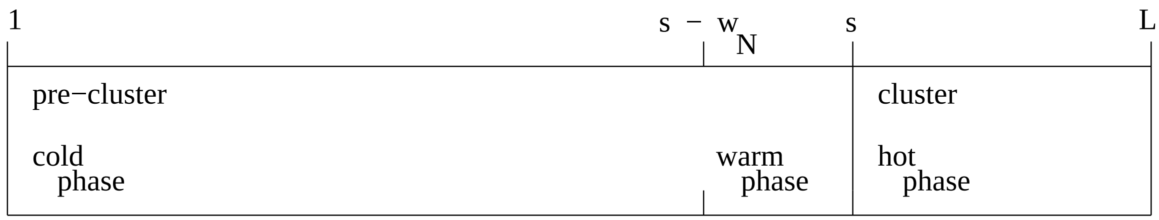

Figure 4.2: Pre-cluster–cluster pair as the discrete interval $[1, L]$; since $N \times 100\%$ of all reuse latency measurements have reuse latencies that fall within the interval $[1, w_N]$, for large enough N, beginning warm up w_N instructions prior to instruction s (which borders the sample cluster) will accurately warm up state.

simultaneously. Finally, there is the unsolved problem of *a priori* selection of tuning variables α and β. MRRL is a more flexible solution developed precisely for rapid pre-cluster warm up. Rather than attempt to touch some proportion of cache blocks, MRRL measures the "amount of temporal locality" and uses this to touch specifically *those blocks that are useful for warming up state prior to the sample clusters.*

Memory reference reuse latency refers to the elapsed time between a reference to some memory address M$[A]$ and the next reference to M$[A]$, where "time" is measured in the number of completed instructions. For the following discussion, consider the instructions in a pre-cluster–cluster pair as bijectively mapped to the discrete interval

$[1, L]$, such that instruction$_1 \mapsto 1$, instruction$_2 \mapsto 2$, ..., instruction$_L \mapsto L$ as pictured along the top line of Figure 4.2. Imagine further, that this interval is partitioned into $n \ll L$ mutually-exclusive buckets whose union is exactly $[1, L]$, as depicted along the top of Figure 4.1. Furthermore, let bucket$_i$ ($i \in \{1, 2, ..., n\}$) represent the interval subset $[a, b]$ for $a \geq 1$ and $b \leq L$. Figure 4.1's bucket$_1$ has the tallest bar, indicating that the majority of consecutive reference accesses have reuse latencies that fall within the interval subset $[1, b]$. In other words, most consecutive reference accesses occur within a *short* time of each other, precisely as one would expect according to the principle of temporal locality. Vanishingly few references occupy bucket$_i$ for increasing i.

I measured MRRLs for each pre-cluster–cluster pair of each benchmark using custom-made MRRL profiling software (see Chapter 6). As the profiler simulates each pre-cluster–cluster pair, the profiling software maintains several associative arrays of memory reference addresses, M$[A]$—one for the instruction stream, one for the data stream, and one for the stream of branch instructions. Each element of the array is logically time-stamped with the number of instructions executed as of the currently simulating memory or branch instruction; if a previously-encountered address is reaccessed, the difference of the previous timestamp and the current number of executed instructions is temporarily stored as $\delta insn$. These $\delta insn$ are used to concurrently build a reuse latency histogram by incrementing the counter associated with the bucket$_i$ that bounds $\delta insn$. When each pre-cluster–cluster section concludes, the profiler outputs the $\delta insn$ histogram. These histograms contain the complete memory reference reuse latency profile for each pre-cluster–cluster pair.

Each histogram gives the count of references for which the number of elapsed instructions between successive accesses lies within the interval subset bucket_i for all n buckets. That is, since bucket_i represents the interval subset $[a, b]$, if $\text{bucket}_i = C$, then it occurred C times that a pair of references to a memory reference address $\text{M}[A]$ were referenced as few as a or as many as b instructions apart. From this, one can calculate the percentage of reaccesses having reuse latencies that fall within an arbitrary union of interval subsets represented by the bucket_is. Of particular interest is the percentage of reaccesses having latencies within the interval $[1, w_N]$, where $N \in [0, 1]$ and w_N is the upper bound on one of the bucket_i intervals. If N corresponds to a percentage of reuse latency measurements, then $100N\%$ of all reuse latency measurements occur in fewer than w_N instructions. In Figure 4.1 $N \times 100\%$ of all references have reuse latencies that fall within the interval $[1, w_N]$. In other words, at the N-th percentile (of reuse latency measurements) *the amount of temporal locality is* w_N *instructions.*

Not surprisingly, the MRRL histograms invariably tell the same story when plotted regardless of the benchmark profiled: A far greater number of references are revisited a small number of instructions after their most recent access *i.e.*, the histogram bucket with the largest population was always bucket_1. (Histograms of MRRL profiles of actual SPEC CPU2000 benchmarks are located in Appendix C.) The more instructions that complete between an access to $\text{M}[A]$ and the beginning of a cluster, the less likely $\text{M}[A]$ is to be accessed again during the cluster. This is exactly as I had expected, in light of previously explored concepts [17, 27, 72].

Quantification of temporal locality is the basis for deciding the length of the warm phase. Let $w_N \mapsto$ bucket$_i$ mean that the i-th partition bucket of the $[1, L]$ interval is upper-bounded at $L - w_N$. This means simply, that of all the reference reaccesses in the current pre-cluster–cluster pair, $N \times$ 100% have reuse latencies of w_N or fewer instructions.

Armed with this knowledge, simulation can begin; if the length of the pre-cluster phase is s instructions, then the warm phase will begin after $s - w_N$ cold phase instructions have completed. By engaging warm up w_N instructions prior to the pre-cluster–cluster boundary, for large enough N [1] the overwhelming majority of addresses M$[A]$ that will be accessed during the eminent sample cluster will have been initialized. I argue that if $N \times 100\%$ of references require only w_N instructions between successive accesses, then it is pointless to model the pre-cluster cache and branch predictor interactions that occur more than w_N instructions before the cluster.

Thus, the steps of the MRRL sampled microarchitecture simulation acceleration technique are:

1. First, the user selects the locations of the cycle-accurate sample clusters within the benchmark; by corollary pre-cluster regions are selected simultaneously.

2. The user next profiles the benchmark to characterize, for each pre-cluster–cluster pair, the reuse latency of all references that occur. As this profile data

[1] A discussion of "large enough" N appears in Chapter 5.

is valid for any cache and branch predictor configuration, this is a one-time cost for each benchmark sample.

3. Simulations can then be run in aggressive cold mode, simulating only architected state. At w_N instructions prior to the cycle-accurate sample cluster, the simulator begins modeling interactions with the cache hierarchy and branch predictor (*i.e.*, warm mode). Once the cluster is reached, the cache(s) and branch predictor will contain accurate state and cycle-accurate simulation begins (*i.e.*, hot mode). Repeat three-phase cold–warm–hot simulation for each pre-cluster–cluster pair.

4.1 Improvements over *MSEwarmup*

At the beginning of the chapter, I claimed that MRRL has superior flexibility and applicability when compared to *MSEwarmup*. The reasons for this include:

1. **Does not require probability computations.** Even though I was able to derive a tractable approximation to the MSE formula, these calculations were still time-consuming because of the lack of a closed-form solution for m (the required number of unique references to achieve probability p of touching the stated proportion of cache blocks). Instead, I was forced to iterate to the correct value of m. MRRL makes no such calculations, eliminating this overhead altogether.

2. **Utilizes fixed block-width granularity MRRLprofiles that can be used with any cache block configuration.** The MRRL profiler always profiles

with word-width granularity. That is, the MRRL profiler always discards the 2 least significant bits as these contain the byte offset within each 32-bit word address. Since MRRL does not need to calculate probabilities based on the number of unique references, the profiler maintains a single definition of "unique." Uniqueness was critically important to MSE because individual bytes and words are not manipulated inside a cache; rather, whole blocks are fetched, invalidated, and evicted. Block-width dependency would require that a benchmark be profiled multiple times to obtain separate measurements if two or more regions of the cache hierarchy used different block widths. This is exactly the case with the MIPS R10000 [40], which has 64-byte L1 instruction-cache blocks, and 32-byte L1 data cache blocks. MRRL can service an R10000-like cache organization from a single profile.

3. **Applicable to warming up branch predictors.** Before discussing *MSE-warmup*'s short-comings, I will briefly review hardware dynamic branch prediction. While there are some variations on the theme, Hennessy and Patterson describes the canonical *branch prediction buffer* (BPB) [21]. The BPB is a small, special-purpose tagless cache of 2^n entries, accessed during instruction fetch, indexed by the n low-order program counter bits. The BPB foregos tags for access rapidity; occasionally therefore, multiple branches will have the same n low-order bits and collide within the BPB, but the likelihood of collisions is mitigated by the fact that branches usually do not constitute a majority of the dynamic instruction stream [21] and can be further alleviated by increasing

the number of BPB entries. Each of the 2^n BPB elements is a 2-bit saturating counter, meaning that when incremented, their values never go above 11_2, and when decremented, their values never go below 00_2. When accessed, the BPB returns the high-order bit from the indexed element. Once the corresponding instruction is decoded, if it is discovered to be a branch, the fetch engine will update the program counter according to this *prediction bit*. A 1 denotes that the branch is predicted *taken* and fetching will begin from the target address; a 0 denotes that the branch is predicted *not taken* and sequential fetching will resume. If when the branch finally executes, it is taken, the BPB responds by incrementing the corresponding saturating counter; if not taken, the corresponding element is decremented. In this way, the BPB is "trained" to a certain branching behavior.

One variation of the BPB described in Hennessy and Patterson is the *two-level* branch prediction buffer (2LBPB). As branches execute, a 2LBPB records the taken–not-taken history of the n most recent branches in a shift register. If a branch is taken, the register shifts a 1 into the least significant position; a 0 is likewise inserted if the branch is not taken. The n-bit pattern of taken–not-taken history is used to index into a table of 2^n 2-bit saturating counters. As instructions are fetched and decoded, predictions are made based on the high-order bit from these saturating counters in the same way as the original BPB. Succinctly stated, BPBs predict from per-branch history, while 2LBPBs predict from branch history *globally*, among all branches. This global branch history scheme divorces branch prediction from the temporal locality of branch reference. Thus, it is difficult to imagine a technique for analytically estimating some reduced amount of warm up necessary to establish accurate 2LBPB state. Such a warm up technique would have to correctly initialize the "important" entries in the table of saturating counters, and make its determination of importance independent of information contained in the branch addresses. (This may be an interesting avenue for future research, but I doubt that the development of such a technique would be possible. I additionally doubt that such research would be worthwhile in light of prior research [7, 8] that suggests branch predictor warm up can be accomplished in a trivial number of instructions.) Because of the complications imposed by global history-based prediction, the following discussion of MRRL's applicability to branch prediction uses Hennessy and Patterson's baseline BPB.

MSEwarmup applies MSE in a brute-force attempt to touch a specific proportion of cache blocks with some user-chosen probability. Unfortunately, it is precisely *MSEwarmup*'s brute-force approach that makes it difficult to use in branch predictor warm up. To use MSE to warm up a BPB, one would have to first calculate the MSE-prescribed m, for the number of BPB entries. (The number of BPB entries is analogous to N—the number of cache sets.) This becomes immediately problematic when one considers the immense size of some branch prediction buffers; the UltraSPARC-III [55], for instance, contains 16K entries. Recall from Chapter 3, that to touch N entries with probability p = 99.9%, a good estimate for m is $16N$, which in this case yields: m = 16(16384) = 262144. In other words, to touch all 16K entries with 99.9% certainty, the simulator would have to warm up enough instructions to witness 262,144 *unique* branches—a daunting request for even the largest pre-cluster phases! In fact, because many programs tend to spend much of their execution time in loops, finding 262,144 unique *instructions* in a given pre-cluster region occurs infrequently. Furthermore, in the absence of a sound technique to accurately calculate α (see Chapter 3), if the pre-cluster instructions contain fewer than the MSE-prescribed m unique references, MSE degenerates to *full-warmup*, completely trading away speed up for accuracy. Hence, *MSEwarmup* is not useful for accelerating branch predictor warm up.

By profiling the reuse latency characteristics of the branch address stream however, MRRL can determine the maximum reuse latency, MRRL_{max} (*i.e.*, $\text{MRRL}_{1.000}$, or MRRL at $N \times 100\% = 100\%$), among branches. Since reuse latency is the count of completed instructions between consecutive accesses to a given branch, MRRL_{max} is the count of instructions between consecutive accesses to the branch that took the longest to revisit. This is an immediate improvement over *MSEwarmup*. Whereas *MSEwarmup* strives to model as many instructions as contain m unique references—which may ultimately degenerate to *fullwarmup*—MRRL can bound the warm up phase by MRRL_{max} instructions, which (according to the MRRL profile plots in Appendix C) usually does not encompass all pre-cluster instructions.

Furthermore, the principle of temporal locality also applies to branch instructions: recently accessed branches will likely be accessed again in the near future. This accounts for the large amount of time programs spend in loops and is verified by Thiébaut [66] who showed that changes in the program counter value can be reliably modeled as *recurrent* fractal random walks on a one-dimensional lattice (see Chapter 2). Because recurrent fractal random walks are much more likely to make short jumps (either forward or backward) between lattice cells than long jumps, recurrent walks tend to stay among a limited grouping of cells for a long time before moving to a new grouping. More concretely, main memory emulates the infinite one-dimensional lattice; its uniquely addressable units (*i.e.*, bytes or words) are the cells of the lattice; and a recurrent walk occurs when a region of code is repeatedly revisited (as by looping) before moving to a different region of the code. By bounding the warm phase by the profiled MRRL_{max} among branches and given that the same branches will likely be visited again and again, MRRL-enabled simulation is able to warm up the BPB accurately, training its entries in less time than *fullwarmup*. (Very short branch predictor training times are corroborated by Co and Skadron [7] who show that dynamic branch predictors can be accurately warmed up in as few as 128K instructions.)

4. **Directly applicable to any depth of cache hierarchy, regardless of unified levels (*e.g.*, unified L2/L3).** The critically important uniformity assumption, so easily verified in the L1 caches, does not necessarily hold in the

usually much larger L2 cache. Furthermore, L2s tend to be unified. Another, more subtle assumption underling *MSEwarmup* is that the MSE formula is being applied to a stream containing one type of reference: instruction fetches or data loads and stores. MRRL does not impose these assumptions.

5. **Unaffected by unique reference distribution or density.** Consider the case where the overwhelming majority of unique memory references are first accessed in an intense burst of start-up activity very early during the pre-cluster instructions. Immediately ensuing is a steady-state period where these references' addresses are reaccessed again and again. Assuming m unique references are available among the pre-cluster instructions, this front-loading of unique references diminishes MSE's ability to speed up the simulation. Since most of the unique references occur at the beginning of the pre-cluster instructions rather than near the end, t will have to be chosen near the beginning of the pre-cluster instructions to accommodate m; this scenario is depicted in Figure 4.3. A similar problem occurs when unique references are sparsely distributed throughout the pre-cluster instructions; this flat-loaded scenario is depicted in Figure 4.4. In summary, while MSE's accuracy is sound regardless of the distribution of unique references, MSE's ability to speed up a simulation hinges upon a suitable number of unique references occurring *toward the end of the pre-cluster period*; see Figure 4.5 for an illustration of such a back-loaded pre-cluster period.

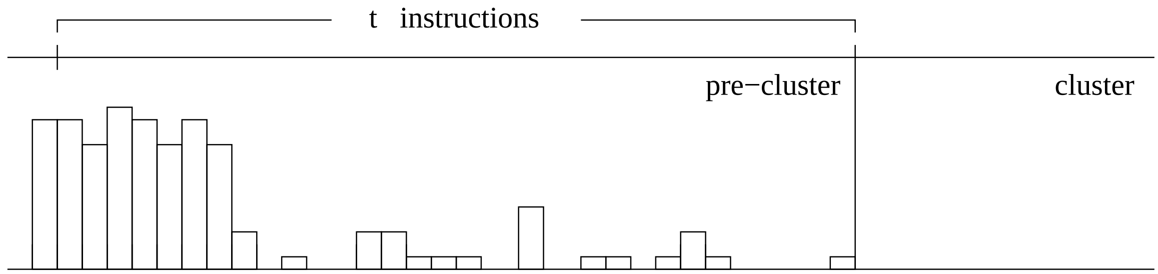

Figure 4.3: Front-loaded pre-cluster contains a burst of first references to unique addresses very early during the pre-cluster period, followed by a sparse population of uniques. t therefore, encapsulates most of the pre-cluster period.

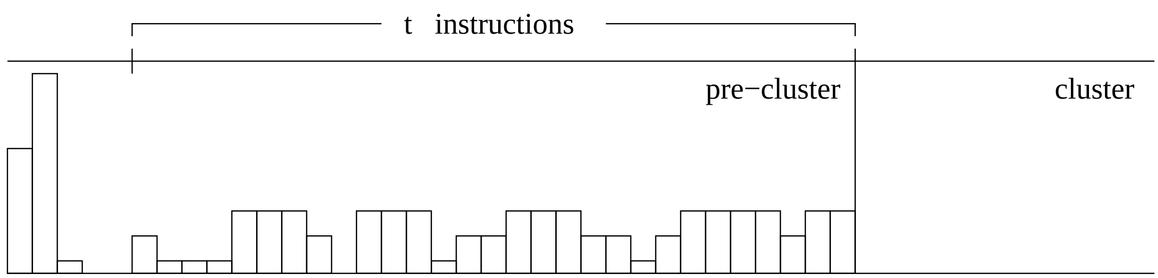

Figure 4.4: Flat-loaded pre-cluster contains sparse, but steady appearance of first references to unique addresses throughout the pre-cluster period. t therefore, encapsulates most of the pre-cluster period.

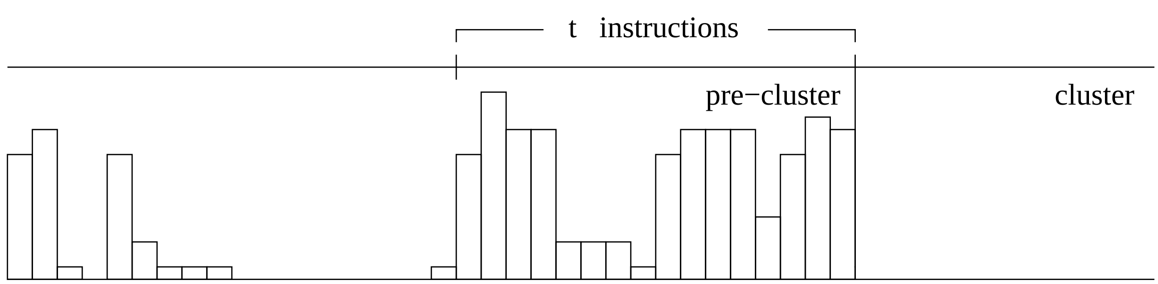

Figure 4.5: Back-loaded pre-cluster contains a burst of first references to unique addresses very late during the pre-cluster period. t therefore, requires only a small portion of the pre-cluster period.

Because MRRL is able to immediately bound the amount of warm up by MRRL_{max}, its ability to accelerate warm up does not depend upon the demography of unique references, and naturally avoids the problems of front-loading and flat-loading entirely. Instead, MRRL warms up instructions according to a certain percentile of reuse latency measurements. Most reference addresses are revisited a small number of instructions after their most recent access (temporal locality); this leads to a clear majority of very short reuse latency measurements. Hence, very high percentiles of reuse latency measurements have a very small number of inter-reference instructions, which lead to the conclusion that: since $N \times 100\%$ of memory reference addresses require only w_N instructions between consecutive accesses, then warming up more than w_N pre-cluster instructions becomes increasingly less useful.

6. **Does not require α or β.** The large capacity of L2s (and even larger L3s) typically preclude their being completely filled, leaving many L2 cache blocks untouched. To accommodate this, MSE introduced the variables α, $\beta \in (0, 1]$, which specify that only a fraction of the sets and a fraction of blocks per set need to be touched. I was unable to find any straight-forward mechanism to determine good α and β *a priori*, which would have forced me to either guess or let $\alpha = \beta = 1$. The former may generate inaccurate results. The latter, due to a dearth of unique references, would usually revert to *fullwarmup* which preserves accuracy, but fails to accelerate warm up.

4.2 Evolution of MRRL from MSE & *MSEwarmup*

Previous research defines any valid cache block whose data will not be reaccessed at least once more before the block is invalidated, evicted and refilled as *dead*; any cache block that is not dead is *live*. A substantial amount of work has been done in the area of analytical characterization of memory reference behavior [11, 62, 66, 67, 68], which has led to a technique for exploring live–dead cache block measurement [39]. Other work adapts dynamic branch prediction to generate informed guesses when a cache block is dead [33].

It might seem that I could adapt this research to develop a technique to predict from the profiles, those cache blocks that would likely remain untouched at the conclusion of the pre-cluster instructions, allowing me to calculate the MSE tuning variables. α would clearly be the quotient of touched sets to total sets. Calculating β would be similar, but not as straight-forward. Should it be calculated as a quotient of the maximum number of touched blocks of all sets? as the average number of touched blocks of all sets? Both these approaches are immediately problematic. If for instance, I chose to calculate β as the quotient of the maximum number of touched blocks inside a set, I could very easily arrive at a situation where one or more sets had all their blocks touched. This would make $\beta = 1$ and is of no help at all. Calculating an average is still unappealing because averages tend to mask relevant trends, making β untrustworthy.

As I wrestled with these issues, the notion occurred to me that I need only ensure that those lines that would be accessed *during the sample clusters*—live and dead—

would be in the cache during the clusters. (In fact, merely ensuring that the cache contains live data would be tremendously incorrect. Accurately modeling cache behavior demands that I place the cache into a state that accurately represents or very closely approximates the state that would have occurred had *fullwarmup* been used. Accordingly, any warm up technique has a responsibility to ensure that dead lines and untouched lines are also accurately represented at the beginning of each cluster.) Further deliberation yielded the insight that measuring reuse latency trends would betray a technique for efficiently and effectively warming up cache state at all levels of the hierarchy without complex, time-consuming calculations and with far fewer assumptions than imposed by MSE.

In light of this insight, I quickly abandoned my exploration of α, β, and live–dead analysis in favor of the more flexible, available MRRL technique. Hence, although MRRL cannot displace MSE's quantitative occupancy assessment contribution, it is the general-purpose replacement for *MSEwarmup*.

Chapter 5

Experimental Methodology and Results

Multiple sets of experiments were performed during this research, first to verify initial hypotheses, and finally to gather actual performance data. In the first set of experiments, I sought to establish the validity and viability of MSE as a tool for quantifiably reasoning about cache occupancy by using the pre-cluster intervals described by Skadron *et al.* [54] for the SPECInt95 [58] benchmarks. In the second set, I verified MSE's flexibility by applying *MSEwarmup* to the multiple-cluster, uniform sampling simulation technique discussed by Haskins and Skadron [17]; the benchmarks for these experiments come from the more up-to-date SPEC CPU2000 [57] suite. In the third set, I demonstrate MRRL's speed, accuracy and superior availability by using the same multiple-cluster, uniform sampling strategy on a simulated microarchitecture that includes a unified L2 cache and a dynamic branch predictor.

The fourth set of experiments applies MRRL to SPEC CPU2000 samples given by Sherwood *et al.* [51], and the final set tests MRRL's speed and accuracy in random cluster sampling, discussed by Conte *et al.* [8].

In all cases, the benchmarks were simulated using their respective reference inputs supplied in the SPECInt95 and SPEC CPU2000 suites. The tools I use in my research were custom-built within the framework provided by the SimpleScalar [4] software suite. These include *sim-safe*, a functional simulator that models purely architected state; *sim-cache*, a multi-level cache hierarchy simulator; *sim-mrrlprofile*, an MSE/MRRL profiling tool built from *sim-safe*; *sim-inorder*, a 6-stage, in-order issue processor core simulator built from *sim-cache*; and *sim-outorder_mrrl*, an out-of-order issue processor core simulator (built from *sim-outorder*) extended to perform three-phase, cold–warm–hot sampling and MSE/MRRL warm up techniques. I also wrote software to perform the MSE calculations according to the MSE direct-mapped approximation described in Chapter 3 that returns a warm phase interval t, given N, a, p and MSEprofile data for a benchmark–input pair.

The data discussed in this chapter were gathered using the steps enumerated in Chapter 3 and Chapter 4. All benchmark binaries were compiled into the Alpha AXP instruction set and statically linked so that the simulations see all user-space program behavior. For each set of experiments, the first step was to execute the one-time profiling pass for each benchmark–input pair. Then, using the MSE/MRRL software in conjunction with the MSEprofile/MRRLprofile data, I found for each cache (and for MRRL, branch predictor configuration), an appropriate pre-cluster

	N	m	Δm
$p = 99.0\%$	512	5544	-32.32%
	1024	11803	-28.02%
	2048	25031	-23.61%
	4096	52906	-19.27%
	N	m	Δm
$p = 95.0\%$	512	4710	-42.50%
	1024	10135	-38.14%
	2048	21693	-33.80%
	4096	46230	-29.46%

Table 5.1: m summary for $p = 99.0\%$ and $p = 95.0\%$ compared to a baseline of $p = 99.9\%$.

point to engage warm up.

5.1 *MSEwarmup*: Single-large-cluster Samples

As previously stated, the first set of experiments sought to experimentally verify the mathematical principles underlying MSE. To demonstrate that MSE is an effective means for reasoning about cache capacity, the first set of experiments used *MSEwarmup* to decide the number of instructions prior to the clusters described by Skadron *et al.* [54] to begin warm up such that all cache blocks would be touched.

I chose my baseline probability of accurate warm up, p, to be 99.9%, and also tested *MSEwarmup* for $p \in \{99.0\%, 95.0\%\}$. Table 5.1 shows for direct-mapped caches (with number of sets, $N \in \{512, 1024, 2048, 4096\}$), the necessary m for lower probabilities of accurate warm up 99.0% and 95.0%, and gives the change in m relative to the m required for 99.9%.

The experiments used *sim-cache* to model direct-mapped caches, fast-forwarding past the cold-phase instructions, and engaging cache modeling at the MSE-prescribed t instructions prior to each benchmark's sample cluster. Once the sample cluster is reached, if the number of cache blocks touched is at least N (I assume $\alpha = 1$), the experiment was successful, because *MSEwarmup* met its objective of touching all N cache blocks.

With probability of accurate warm up chosen to be $p = 99.9\%$, all N sets were indeed touched after the MSE-prescribed t warm up instructions for all benchmarks tested. When the probability was adjusted to $p = 99.0\%$ and $p = 95.0\%$, the MSE-prescribed t warm up instructions usually touched all N sets. When it did not however, only a very small number of sets (fewer than 10) were excluded. Of particular interest in these experiments is the size of the warm phase as a percentage of all pre-cluster instructions. Tables 5.2 through 5.5 give the percentage of N sets touched and the percentage of the pre-cluster interval that was warmed up for each benchmark.

For all benchmarks except *go*, fewer than 16% of the pre-cluster instructions suffices to touch all or minutely fewer than N sets. *go*'s higher percentage of pre-cluster warm up for $N = 2048$ with $p = 99.9\%$ and $N = 4096$ for all p, is due to its front-loaded unique reference distribution (see Figure 4.3). Figure 5.1 illustrates the unique reference address distribution for *go*. Each point on the x-axis denotes some number (in hundreds) of unique reference addresses; each corresponding y (along the logarithmic y-axis) gives the number of instructions that must be completed in order to see $100x$ unique reference addresses (*i.e.*, each y gives the size of the cold phase necessary to access $100x$ uniques).

	$p = 99.9\%$		$p = 99.0\%$		$p = 95.0\%$	
benchmark	% of N	% of pre-cluster	% of N	% of pre-cluster	% of N	% of pre-cluster
compress	100%	0.11%	100%	0.08%	100%	0.07%
gcc	100%	4.16%	100%	3.28%	100%	3.05%
go	100%	0.08%	100%	0.08%	100%	0.07%
ijpeg	100%	0.28%	100%	0.12%	100%	0.12%
m88ksim	100%	1.43%	100%	1.06%	100%	0.05%
perl	100%	13.89%	99.61%	12.42%	99.02%	10.70%

Table 5.2: SPECInt95 benchmark summary for $N = 512$, $p \in \{99.9\%,99.0\%,95.0\%\}$.

	$p = 99.9\%$		$p = 99.0\%$		$p = 95.0\%$	
benchmark	% of N	% of pre-cluster	% of N	% of pre-cluster	% of N	% of pre-cluster
compress	100%	0.25%	100%	0.17%	100%	0.15%
gcc	100%	5.61%	100%	4.39%	100%	4.33%
go	100%	1.98%	100%	0.35%	100%	0.10%
ijpeg	100%	0.51%	100%	0.32%	100%	0.30%
m88ksim	100%	2.51%	100%	1.80%	100%	0.12%
perl	100%	14.09%	99.12%	13.95%	99.84%	13.91%

Table 5.3: SPECInt95 benchmark summary for $N = 1024$, $p \in \{99.9\%,99.0\%,95.0\%\}$.

	$p = 99.9\%$		$p = 99.0\%$		$p = 95.0\%$	
benchmark	% of N	% of pre-cluster	% of N	% of pre-cluster	% of N	% of pre-cluster
compress	100%	0.53%	100%	0.39%	100%	0.33%
gcc	100%	9.88%	100%	8.16%	100%	6.44%
go	100%	31.90%	100%	4.79%	100%	2.53%
ijpeg	100%	0.94%	100%	0.74%	100%	0.69%
m88ksim	100%	5.03%	100%	3.95%	100%	3.59%
perl	100%	14.59%	100%	14.37%	100%	14.27%

Table 5.4: SPECInt95 benchmark summary for $N = 2048$, $p \in \{99.9\%,99.0\%,95.0\%\}$.

	$p = 99.9\%$		$p = 99.0\%$		$p = 95.0\%$	
benchmark	% of N	% of pre-cluster	% of N	% of pre-cluster	% of N	% of pre-cluster
compress	100%	0.96%	100%	0.77%	100%	0.69%
gcc	100%	15.55%	99.98%	13.83%	99.98%	12.10%
go	100%	99.90%	100%	99.89%	100%	99.74%
ijpeg	100%	1.96%	100%	1.63%	100%	1.40%
m88ksim	100%	10.05%	100%	7.90%	100%	7.16%
perl	100%	15.57%	100%	15.21%	100%	15.01%

Table 5.5: SPECInt95 benchmark summary for $N = 4096$, $p \in \{99.9\%,99.0\%,95.0\%\}$.

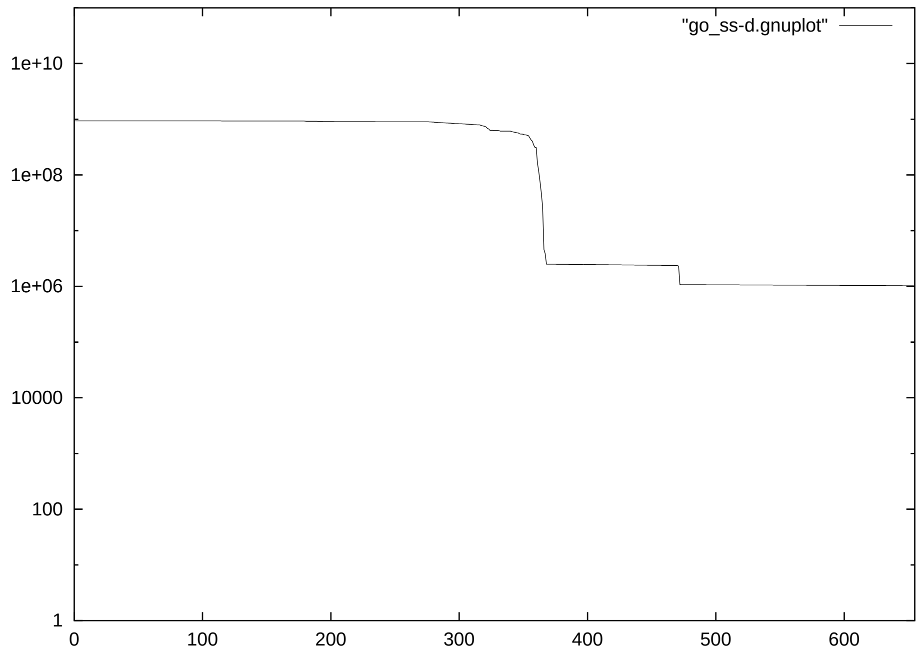

Figure 5.1: MSE unique reference plot for *go*. The x-axis gives the MSE-prescribed m (in hundreds of unique references); the y-axis gives the number of cold phase instructions.

The pre-cluster period described by Skadron *et al.* [54] for *go* consists of the first 925 million instructions from the start of execution. Consider $N = 512$ and $p = 99.9\%$ for *go* in Table 5.2; in this experiment, the warm phase is comprised of the last 0.08% of the pre-cluster instruction stream. Since the MSE-prescribed m for $N = 512$ with $p = 99.9\%$ is 8,192, the corresponding point on the Figure 5.1 curve—at roughly $x = 82$—is very close to $y = 925$ million; this indicates a very long cold phase of slightly fewer than 925 million instructions suffixed by a very brief warm phase. Notice the very sharp drop in the curve for $x > 350$, of nearly three orders of magnitude from $y = 10^9$ to $y = 10^6$. Since the y-axis gives the cold phase duration, this large drop implies a marked increase in the warm phase duration for those *go* simulations whose

MSE-prescribed m is greater than 35,000 unique references (*i.e.*, for N = 4,096, with $m_{p=95.0\%}$ = 46,230, $m_{p=99.0\%}$ = 52,906, and $m_{p=99.9\%}$ = 65,536). These experiments had to engage warm up very early (after roughly 10^6 instructions) in the pre-cluster instruction stream to encounter the MSE-prescribed m unique memory references. This accounts for the 99%+ amount of the pre-cluster instruction stream occupied by the warm phases shown in Table 5.5.

Unique memory reference address plots for all the SPECInt95 benchmarks used in this dissertation are located in Appendix B. Notice that only *go* drops off so sharply while the others fall only very slightly as x increases. For these other benchmarks therefore, the cold phase contains of the majority of the pre-cluster instructions.

5.2 *MSEwarmup*: Uniform Multiple-cluster Samples

Having vindicated MSE through experimentation with *MSEwarmup*, the next step was to apply *MSEwarmup* to uniform multiple-cluster sample simulation. In this second set of experiments, my objective was to demonstrate that *MSEwarmup* can obtain simulation that closely mimics *fullwarmup*, but in less time. As stated in Chapter 1, simulation accuracy is predicated upon successfully defeating the cold-start bias. This is accomplished by establishing accurate simulated state prior to actual data gathering that occurs during the cycle-accurate cluster simulations. Modeling all pre-cluster cache interactions makes *fullwarmup* impervious to cold-start bias since cache state is perfectly maintained. The results of the previous set of experiments indicate that *MSEwarmup* simulations should execute substantially faster than *full-*

warmup because in general, only a small proportion of pre-cluster instructions were necessary to touch all cache blocks. If the instruction throughput (IPC) obtained through *MSEwarmup* simulation is close to that obtained through *fullwarmup* simulation and the *MSEwarmup* simulations execute in less time than *fullwarmup*, then *MSEwarmup* is a sound cache warm up strategy.

These experiments were conducted using *sim-inorder*—a custom-built cycle-accurate processor simulator that models a 6-stage, 4-way, in-order issue pipeline with dynamic branch prediction. The L1 instruction- and data-cache were configured to be 2-way associative with 1,024 sets and 32-byte blocks for a total capacity of 64 kilobytes apiece. Each uniform multiple-cluster sample measured instruction throughput for the first 25 billion instructions from each benchmark. Each cluster was spanned 1 million instructions and preceded by 499 million pre-cluster instructions. To provide a thorough test of *MSEwarmup*, both the instruction- and data-cache were flushed at the conclusion of each cycle-accurate cluster. Doing so actually makes *MSEwarmup*'s task harder by rendering the simulations unable to take advantage of previously-fetched blocks that would have otherwise remained in the cache. This is in contrast to the random cluster sampling method proposed by Conte *et al.* [8] which opts instead to maintain "stale" state between samples. I tested two probabilities of successful warm up: $p = 99.9\%$ and $p = 95.0\%$, and conservatively sought t prior to each full-detail cluster sufficient to touch all N sets at least twice ($\alpha = 1$, $a = 2$) using the direct-mapped approximation described in the Chapter 3.

Critical to these experiments was the amount of cache warm up prior to making

IPC measurements during the cycle-accurate clusters. Four warm up strategies were tested: *shortwarmup*, $MSEwarmup_{95.0\%}$, $MSEwarmup_{99.9\%}$ and *fullwarmup*. *shortwarmup* warms up cache state (*i.e.*, models cache interactions) for 7,000 instructions [8] prior to cycle-accurate simulation; $MSEwarmup_{95.0\%}$ and $MSEwarmup_{99.9\%}$ warm up cache state for their respective MSE-prescribed t instructions prior to cycle-accurate simulation; and *fullwarmup* warms up cache state during the entire pre-cluster period.

I measured the goodness of *MSEwarmup* by two metrics. The first is IPC accuracy measured as the percent-error between *MSEwarmup* and *fullwarmup* calculated:

$$\frac{IPC_{MSEwarmup} - IPC_{fullwarmup}}{IPC_{fullwarmup}}$$

Table 5.6 (and Figure 5.2) shows the *MSEwarmup* IPC accuracy results and contrasts them with the IPC accuracies obtained through *shortwarmup*. The second metric is the fraction of *fullwarmup* simulation running time calculated: $100\% \cdot (\frac{t_{MSEwarmup}}{t_{fullwarmup}})$. Table 5.7 (and Figure 5.3) gives these results and contrasts them against the fraction of *fullwarmup* simulation time necessary for *shortwarmup*. Both metrics are tightly coupled; the latter is worthless without the former. In other words, reducing simulation time is only useful if it is done while simultaneously preserving accuracy.

From Table 5.6 it is clear that the MSE-prescribed warm up yields superior IPC precision relative to *shortwarmup* in general. Of fifteen benchmarks, fourteen yielded IPCs that are closer (*i.e.*, have a smaller absolute value percent error relative to *fullwarmup*) than *shortwarmup* for both $p = 99.9\%$ and $p = 95.0\%$. The singular benchmark *applu* was probably adversely affected by the fresh-state approach the

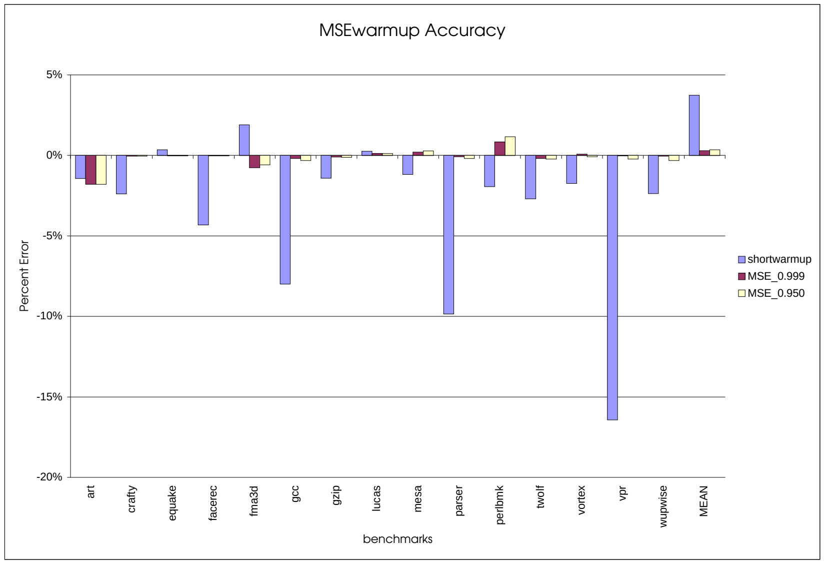

Figure 5.2: *MSEwarmup* cache accuracy relative to *fullwarmup*.

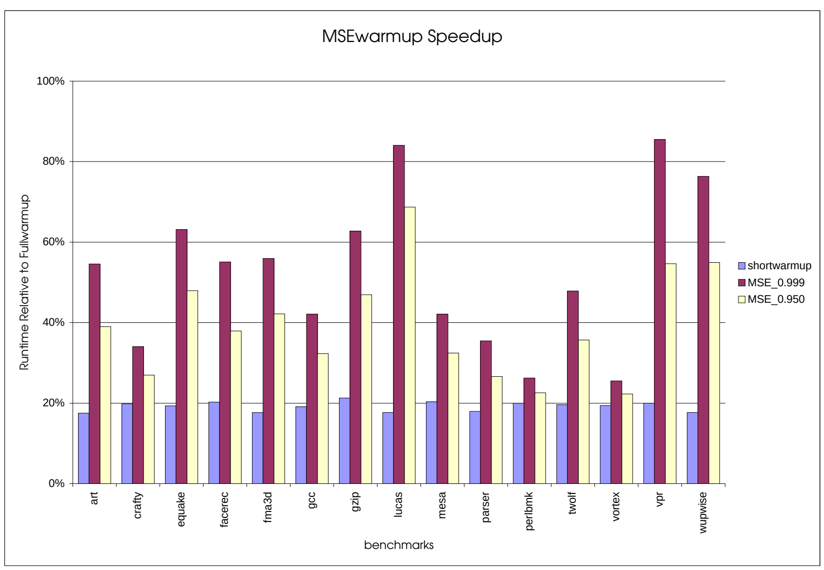

Figure 5.3: *MSEwarmup* speedup relative to *fullwarmup*.

		% error		
benchmark	*fullwarmup*	*shortwarmup*	$MSEwarmup_{99.9\%}$	$MSEwarmup_{95.0\%}$
applu	0.7857	-1.425%	-1.769%	-1.769%
crafty	1.3946	-2.373%	-0.029%	-0.029%
equake	0.6146	0.358%	-0.016%	-0.016%
facerec	1.2042	-4.293%	-0.008%	-0.008%
fma3d	0.8492	1.896%	-0.742%	-0.565%
gcc	1.0665	-7.979%	-0.169%	-0.291%
gzip	1.5224	-1.399%	-0.085%	-0.099%
lucas	0.7439	0.255%	0.121%	0.121%
mesa	1.3797	-1.160%	0.210%	0.275%
parser	1.0851	-9.833%	-0.065%	-0.175%
perlbmk	1.0542	-1.916%	0.844%	1.157%
twolf	1.2008	-2.682%	-0.167%	-0.208%
vortex	1.1118	-1.727%	0.072%	-0.063%
vpr	1.0675	-16.42%	-0.019%	-0.206%
wupwise	0.9783	-2.361%	-0.020%	-0.307%
MEAN		3.738%	0.289%	0.353%

Table 5.6: Result summary for 50-sample simulation IPCs. This table compares the IPC from *fullwarmup* to the percent difference $\frac{(IPC - IPC_{fullwarmup})}{IPC_{fullwarmup}}$ in IPC for both *shortwarmup* and *MSEwarmup*. The mean of percent differences was calculated using their absolute values.

experiments took, flushing the instruction- and data-cache at the conclusion of each cycle-accurate cluster.

Clearly *shortwarmup* is superior in terms of simulation running time, never taking longer than 22% of the time taken by *fullwarmup* simulation. The running times obtained by *MSEwarmup* however, are also smaller than *fullwarmup*, taking only as much as 86% or as little as 26% for $p = 99.9\%$ and as much as 69% or as little as 22% for $p = 95.0\%$ of the time required by *fullwarmup*. These running times reflect the number of pre-cluster instructions that were used for warm up between the cycle-

		% of original running time		
benchmark	*fullwarmup*	*shortwarmup*	$MSEwarmup_{99.9\%}$	$MSEwarmup_{95.0\%}$
applu	26572 sec.	17.53%	54.59%	38.98%
crafty	27175 sec.	19.80%	34.01%	27.00%
equake	27220 sec.	19.35%	63.14%	47.92%
facerec	26190 sec.	20.28%	55.11%	37.93%
fma3d	27591 sec.	17.65%	55.96%	42.17%
gcc	28377 sec.	19.07%	42.10%	32.30%
gzip	27037 sec.	21.24%	62.73%	46.94%
lucas	25739 sec.	17.70%	84.07%	68.73%
mesa	26602 sec.	20.30%	42.14%	32.48%
parser	27735 sec.	17.98%	35.48%	26.59%
perlbmk	27905 sec.	18.98%	26.27%	22.59%
twolf	27967 sec.	19.64%	47.90%	35.67%
vortex	28301 sec.	19.40%	25.49%	22.27%
vpr	28235 sec.	19.96%	85.53%	54.67%
wupwise	26173 sec.	17.66%	76.32%	54.93%
MEAN		19.10%	52.72%	39.41%

Table 5.7: Result summary for 50-sample simulation running times (in seconds). This table compares the running times for *fullwarmup* to the percentage of this time for both *shortwarmup* and *MSEwarmup*.

accurate clusters. In a simulation such as *vpr* (the longest running benchmark for $p = 99.9\%$), the explanation for its nearly 86% measurement is the fact that each 499-million-instruction pre-cluster phase was sparsely populated by *unique* memory references. Therefore, the MSE-prescribed t memory references during the pre-cluster intervals had to be large to capture the m necessary uniques in order to achieve probability p of accurate warm up. In fact, for some of the benchmarks, several of the pre-cluster intervals were so sparse with unique memory references that they did not contain m uniques; for these intervals the *MSEwarmup* spans the entire pre-cluster phase, degenerating to *fullwarmup*. When this occurs, no speed-up over *fullwarmup* is realized for the current pre-cluster–cluster pair, but this is presumably preferable to ad-hoc techniques that may achieve speed-ups at the expense of accuracy.

Compared to $MSEwarmup_{99.9\%}$, *shortwarmup* reduced simulation times by roughly a factor of 2.75 and by roughly a factor of 2.05 compared to $MSEwarmup_{95.0\%}$; on the other hand, $MSEwarmup_{99.9\%}$ yields results that are roughly 12.9 times more accurate than *shortwarmup* and $MSEwarmup_{95.0\%}$ is roughly 10.6 times more accurate (see the MEAN entries of Table 5.6 and Table 5.7). This is an interesting result: A decreased probability of accurate warm up ($p = 95.0\%$) reduces simulation running time, yet in general still achieves a more accurate IPC measurement than *shortwarmup*. As hypothesized, MSE rigorous mathematical approach to determining suitable pre-cluster warm up intervals is more reliable than previous, more ad-hoc methods. This is especially evident when one examines the IPCs achieved by individual benchmarks rather than the average case. Notice that while *shortwarmup* has a mean error of only 3.7%, outlying benchmarks such as *gcc*, *parser* and *vpr* have much higher errors (8%, 9.8% and 16.4%, respectively); this shows that *shortwarmup* cannot be trusted. On the other hand, MSE never achieves an error greater than 1.8%.

5.3 MRRL: Uniform Multiple-cluster Samples

In the previous set of experiments, I used *MSEwarmup* to demonstrate the effectiveness of the three-phase cold–warm–hot simulation strategy for accelerating warm up while preserving simulation accuracy. This set of experiments builds upon those re-

Component	Configuration
Instruction cache	64KB, 2-way, 32B blocks, 1 cycle access latency
Data cache	64KB, 2-way, 32B blocks, 1 cycle access latency
Unified L2 cache	2048KB, 4-way, 32B blocks, 11 cycle access latency
Branch predictor	Hybrid: 4K BiMod & 12-bit GAg, 32-entry RAS

Table 5.8: Simulator cache hierarchy and branch predictor configuration for MRRL experiments.

sults, but uses MRRL to delineate the boundary between the cold and hot phases of simulation. Recall from Chapter 4 however, that MRRL can be used to warm up all levels of the cache hierarchy and a dynamic branch prediction buffer. Hence, these experiments utilize the more sophisticated CPU configuration shown in Table 5.8.

As enumerated in Chapter 4, benchmarks were first partitioned into pre-cluster–cluster pairs and profiled to characterize the reuse latencies of each. Pursuant to this, the warm phase was engaged w_N instructions prior to the cluster. This process was repeated for every benchmark for $N \in \{0.950, 0.990, 0.995, 0.999\}$.

This set of experiments used a multiple-cluster uniform sampling strategy different from the previous set. The previous set simulated a fixed number (50) of 10-million-instruction clusters, uniformly located throughout the first 25 billion dynamic instructions. Rather than limiting the range of the clusters to a fixed subset of the dynamic instruction stream, this set of experiments samples 10% of the end-to-end dynamic instruction stream in uniformly-spaced 10-million-instruction clusters. As before, precisely the same pre-cluster–cluster partition information that was fed to the profiler was also fed to the multiple-cluster simulator.

On their reference input data, the dynamic instruction counts for the SPEC CPU2000 benchmarks are large, usually on the order of 10^{10}. Consequently, these experiments' 10% strategy simulated many more sample clusters than the previous set. This greatly reduced the amount of achievable speed up since under the three-phase cold–warm–hot simulation strategy, acceleration is accomplished by speeding up simulation of *pre-cluster* instructions. This effect can be seen by comparing Table 5.7 and Table 5.10. Notice in Table 5.7 that *shortwarmup*'s running time as a percentage of *fullwarmup* is always less than 25%; compare this to Table 5.10, where *nowarmup*'s percentage of *fullwarmup* running time is greater than 60% on average.

5.3.1 IPC accuracy and speed-up

Table 5.9 (and Figure 5.4) shows the percent error in IPC relative to *fullwarmup* using the MRRL warm up technique and the *nowarmup/stalestale* technique, which—as its name suggests—makes no effort to establish correct state prior to each sample cluster other than recycling state as it appeared at the conclusion of the prior cluster [8]. (For brevity, we shorten *nowarmup/stalestate* to *nowarmup*.) In other words, *nowarmup* experiments did not model any cache or branch predictor interactions prior to the clusters. This makes *nowarmup* susceptible to cold-start bias as is readily seen from the benchmarks *facerec* and *gcc*, and more dramatically from *vpr* and *parser*. Though most benchmarks' *nowarmup* IPC diverges by less than 1% from *fullwarmup*, the benchmark *parser* qualitatively demonstrates the phenomenon of cold-start bias. Accurate simulation is predicated upon establishing an accurate representation of the simulation environment; if the environment is inaccurate, so will be the results of the

		IPC %-error				
benchmark	$\mathrm{IPC}_{fullwarmup}$	*nowarmup*	$\mathrm{MRRL}_{0.950}$	$\mathrm{MRRL}_{0.990}$	$\mathrm{MRRL}_{0.995}$	$\mathrm{MRRL}_{0.999}$
art	2.0708	-0.7920%	-0.1062%	-0.0338%	-0.0048%	0.0000%
crafty	2.3966	-0.0209%	-0.0250%	-0.0250%	-0.0250%	-0.0125%
facerec	1.6675	-2.4048%	-0.1979%	-0.1379%	-0.0960%	-0.0240%
fma3d	1.4186	-0.8248%	-0.4582%	-0.4018%	0.0141%	0.0000%
gcc	1.9937	-2.6433%	-1.2991%	-0.7323%	-0.4414%	0.0752%
gzip	2.1777	-0.1056%	-0.0413%	-0.0276%	-0.0138%	-0.0092%
lucas	0.9627	0.0831%	0.0312%	0.0208%	0.0208%	0.0208%
mesa	2.4695	0.3442%	0.3766%	0.4049%	0.3968%	0.3887%
parser	1.5248	-9.3061%	-6.4861%	-2.4200%	-1.2133%	-0.2689%
perlbmk	1.6350	1.2966%	1.3089%	1.3394%	-0.0979%	-0.0061%
twolf	1.5647	0.0000%	-0.0192%	-0.0128%	-0.0128%	-0.0128%
vortex	2.2447	-0.6593%	-0.4990%	-0.4767%	-0.3742%	-0.1114%
vpr	1.1182	-4.5698%	-2.2447%	-0.5187%	-0.2236%	-0.0179%
wupwise	1.8261	0.4600%	0.0000%	0.0000%	0.0055%	0.0000%
MEAN		1.6793%	0.9352%	0.4680%	0.2100%	0.0677%

Table 5.9: IPC accuracy as %-error relative to *fullwarmup* ($100\% \cdot \frac{IPC_{MRRL_N} - IPC_{fullwarmup}}{IPC_{fullwarmup}}$). Mean values were calculated from the absolute value of the %-error measurements.

simulation. This intuitively establishes the untrustworthiness of insufficient warm up. A quantitative, statistically rigorous demonstration of this untrustworthiness is the topic of 5.6.

Notice from Table 5.9 that MRRL_N shows an increasing trend toward enhanced accuracy (*i.e.*, smaller relative error absolute value) as N increases. This result becomes even more compelling when one observes the trend on the four *nowarmup* experiments highlighted previously. The only exception to the monotone increasing trend among the benchmarks is *mesa*. I speculate that this is the result of destructive interference in the branch predictor which causes *mesa* to enter into one or more of its clusters with suboptimal branch predictor state. Nevertheless, in all cases, $\mathrm{MRRL}_{0.999}$ achieves an error of less than 0.4% deviation from *fullwarmup*; lower values of N were less reliable, but in general, more accurate than *nowarmup*.

Before my discussion of MRRL's ability to accelerate simulation running times, it is important to establish the optimality of *nowarmup*'s runtime relative to *fullwarmup*. Since *nowarmup* does not model any cache or branch predictor interactions prior to the cycle-accurate clusters, the *nowarmup* simulations have no warm phase, only cold and hot. The cold phase simulates in a lower level of detail than the warm phase. (This translates into fewer native instructions per simulated instruction and therefore, more rapid execution.) If the hot phase cannot be changed or removed, then eliminating the warm phase altogether, minimizes execution time to is absolute minimum under the three-phase cold–warm–hot simulation strategy. By corollary, under the same assumption of an indelible hot phase, eliminating the cold phase altogether (as happened when *MSEwarmup* found fewer than m unique pre-cluster references), yields the maximum cold–warm–hot simulation time.

Since *nowarmup* running time is the minimum possible running time it also represents the per-benchmark maximum potential speed-up. The $\%_{nowarmup}$ column from Table 5.10 (and Figure 5.5) shows that these potential speed-ups ranged from 59.83% for *art* to 76.25% for *fma3d*, where these are the percentage of each benchmark's *fullwarmup* running time ($100\% \cdot \frac{t_{nowarmup}}{t_{fullwarmup}}$). All MRRL_N running time percentages shown in Table 5.10 give the percentage of potential speed up that each simulation achieved ($100\% \cdot (1 - \frac{t_{MRRL_N} - t_{nowarmup}}{t_{nowarmup}})$).

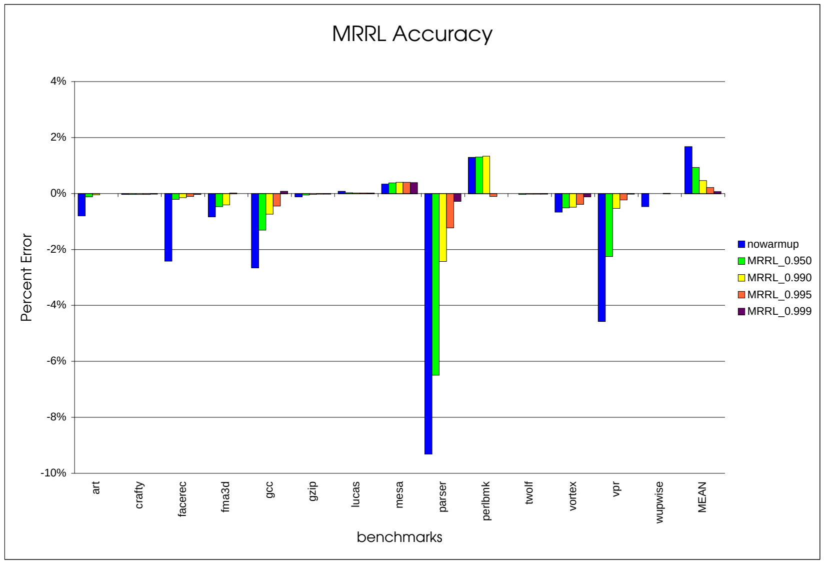

Figure 5.4: MRRL multiple-cluster cache accuracy relative to *fullwarmup*.

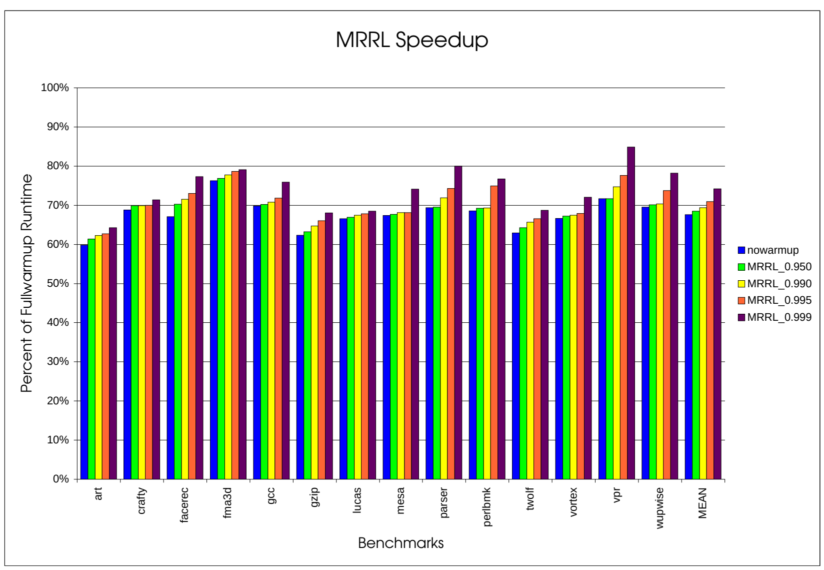

Figure 5.5: MRRL multiple-cluster speedup relative to *fullwarmup*.

benchmark	$t_{fullwarmup}$	$\%_{nowarmup}$	$\%_{MRRL_{0.950}}$	$\%_{MRRL_{0.990}}$	$\%_{MRRL_{0.995}}$	$\%_{MRRL_{0.999}}$
art	2761 sec.	59.83%	97.46%	95.58%	95.16%	92.62%
crafty	109873 sec.	68.76%	98.35%	98.35%	98.28%	96.21%
facerec	114071 sec.	67.06%	95.23%	93.39%	91.17%	84.70%
fma3d	171281 sec.	76.25%	99.14%	98.04%	96.89%	96.24%
gcc	66856 sec.	69.80%	99.46%	98.55%	97.07%	91.23%
gzip	48673 sec.	62.37%	98.59%	96.29%	94.05%	90.86%
lucas	80891 sec.	66.54%	99.36%	98.68%	98.09%	97.07%
mesa	79019 sec.	67.37%	99.52%	98.92%	98.85%	90.00%
parser	327684 sec.	69.39%	99.77%	96.34%	92.96%	84.75%
perlbmk	16636 sec.	68.57%	99.07%	98.95%	90.72%	88.15%
twolf	213200 sec.	62.91%	97.84%	95.65%	94.20%	90.76%
vortex	77808 sec.	66.64%	99.06%	98.78%	98.16%	91.91%
vpr	57293 sec.	71.68%	99.99%	95.71%	91.77%	81.54%
wupwise	192069 sec.	69.55%	99.21%	98.91%	93.92%	87.60%
MEAN			98.72%	97.32%	95.09%	90.26%

Table 5.10: Maximum potential ($\%_{nowarmup}$) acceleration ($100\% \cdot \frac{t_{nowarmup}}{t_{fullwarmup}}$) and achieved percentage of potential ($\%_{MRRL_N}$) running time speed-up ($100\% \cdot (1 - \frac{t_{MRRL_N} - t_{nowarmup}}{t_{nowarmup}})$). Running times measured in seconds.

Observe that for higher percentiles, the amount of achieved potential decreases. This, of course, is due to the fact that higher N increase the size of the warm phase while simultaneously decreasing the cold phase, causing an ever larger proportion of pre-cluster cache and branch predictor interactions to be modeled. In spite of this, achieved potential is still respectable, ranging from 81.54% for *vpr* to 97.07% for *lucas* at $N = 0.999$. These translate into running times of only 84.91% and 68.49%, respectively for each of these benchmarks relative to their *fullwarmup* running times[1]. Thus, for all benchmarks and all percentiles N, running time was reduced by a minimum of 15%.

[1]Percentage of *fullwarmup* $= (1 + (1 - \%_{MRRL_N})) \cdot \%_{nowarmup}$.

5.3.2 Cache and branch predictor accuracy

Although instruction throughput and simulation running times were the primary metrics that were examined to gauge the effectiveness of MRRL, cache and branch predictor performance statistics were also gathered during the simulations. The additional measurements—shown in Table 5.11—help to elucidate the behavior of benchmarks which yielded the most inaccurate IPCs for *nowarmup* and MRRL experiments with a too low value for N. Specifically, this section discusses the benchmarks *parser*, *vpr*, *facerec*, and *gcc*. It is important to first note however, that these extra data were gathered exclusively during the benchmarks' cycle-accurate hot phases of simulation. This was necessary because in general, for different N, in the pre-cluster instructions. By gathering cache and branch predictor statistics exclusively at the intersection of their active lifetimes (that is, exclusively during the hot phase), their performance measurements are guaranteed to be comparable for all N. [Note: MRRL_0 = *nowarmup*.]

MRRL engages the cache and branch predictor warm up at different points depending upon the simulator configuration (*i.e.*, MRRL value of N), per benchmark it was necessary to engage the *measurement* of cache and branch predictor performance only upon the intersection of their active lifetimes: the hot phase. This—unlike taking measurements over the union of the warm and hot phases—guarantees that a benchmark's resultant measurements are comparable among all simulator configurations.

Notice that Table 5.11 only contains cache performance statistics for the unified secondary cache, and omits the primary instruction- and data cache, and branch

		%-error				
benchmark	*fullwarmup*	*nowarmup*	$\mathrm{MRRL}_{0.950}$	$\mathrm{MRRL}_{0.990}$	$\mathrm{MRRL}_{0.995}$	$\mathrm{MRRL}_{0.999}$
parser	0.1468 M/R	53.88%	36.10%	13.08%	6.68%	1.43%
vpr	0.1960 M/R	17.70%	8.27%	1.58%	0.66%	0.05%
facerec	0.3915 M/R	2.71%	0.41%	0.28%	0.20%	0.03%
gcc	0.0419 M/R	25.06%	24.74%	6.21%	3.82%	-0.72%

Table 5.11: Unified second-level cache miss rate %-error relative to *fullwarmup*; M/R = misses per reference.

predictor performance data; these data were simply not interesting. For all N, the 2-way associative, 64KB primary instruction cache produced negligible miss rates of less than 0.0010 misses/reference for all the benchmarks. These consistently low miss rates did not reveal any additional insight. The primary data cache miss rates on the other hand, were not uniformly low, but for all N were very similar, yielding percent differences on the order of 10^{-4}. Just as with the uniformly low instruction cache miss rates, the similarity of the data cache measurements did not reveal any additional information. The branch predictor performance measurements also exhibited strong uniformity[2], in all cases yielding percent differences of less than 0.0207% for direction prediction and less than 0.0106% for address prediction. Once again, this remarkably uniform behavior did not further elucidate the experiments' relative performance,

[2]The consistent accuracy of the branch predictor among multiple simulator configurations is not surprising in light of Conte *et al.*'s work [8], which shows that reliable branch predictor warm up can be achieved by as few as 7,000 warm up instructions; and is further corroborated by Co and Skadron [7] who show (for a modern microprocessor that switches among multiple execution contexts) that a branch predictor can train in as few as 128K instructions. This training duration was negligible relative to the 10-million-instruction clusters.

and they too have been omitted. Thus, I conclude that the key requirement which must be satisfied for accurate warm up and therefore accurate instruction throughput measurements is the miss rate of the unified secondary cache.

Consider the worst offender in terms of IPC accuracy, *parser*, with a -9.3% *nowarmup* percent-error deviation from *fullwarmup*. Recall that the application of MRRL warm up for progressively larger values of N steadily improved the IPC error to -0.3% for N = 0.999. This trend closely mimics the trend shown in Table 5.11, where *nowarmup*'s L2 miss rate deviates by 53.88% from *fullwarmup*'s and decreases with MRRL for progressively larger values for N, culminating at only 1.43% for $N = 0.999$. Precisely the same trend arises for the other benchmarks as well, although seemingly not as pronounced for *facerec*. Notice that *facerec*'s *nowarmup* percent error deviation from *fullwarmup* is only 2.71%. Because the *fullwarmup* miss rate for *facerec* was so high to begin with, however (0.3915 misses/reference), even a modest miss rate increase was manifested as decreased instruction throughput.

5.4 MRRL: Basic Block Distribution Analysis Sampling

Having previously qualitatively and quantitatively established MRRL's soundness as a warm up technique, this set of experiments demonstrates another key objective of my research: flexibility and utility with any sampling regime. Sherwood *et al.* [50, 51] have developed a highly accurate technique for systematically choosing simulation clusters from the a benchmark's end-to-end dynamic instruction stream. When executed in cycle-accurate detail, these *simulation points* yield reliable IPC measurements, while executing in significantly less time than end-to-end cycle-accurate simulation.

For these experiments, the sample clusters are the 100-million-instruction simulation points determined in [51]. For each benchmark–input pair I tested the following warm up methods: *fullwarmup*, *nowarmup*, $\mathrm{MRRL}_{0.999}$ and $\mathrm{MRRL}_{1.000}$. Accuracy results are listed in Table 5.12 (and Figure 5.6). The first column gives the weighted multiple-cluster IPC (calculated according to [51]) for *fullwarmup*; subsequent columns give the percent error deviation from *fullwarmup*. (Sherwood's simulation points [51] and $\mathrm{MRRL}_{0.999}$ warm up points for instructions, data, and branches are listed in Appendix A.)

As expected, MRRL adapted easily to the different pre-cluster–cluster phases prescribed by Sherwood *et al.* [51], never generating an error worse than -1.31% for $N = 0.999$. The highlight of Table 5.12 however, is the pristine performance of $\mathrm{MRRL}_{1.000}$. This unswerving accuracy is very easily explained: Choosing $N = 1.000$ captures the *maximum* memory reference reuse latency *i.e.*, $\mathrm{MRRL}_{1.000} = \mathrm{MRRL}_{max}$. Recall that for some memory address $\mathrm{M}[A]$, the reuse latency is the count of completed instructions between consecutive accesses. MRRL_{max} therefore, is the count of completed instructions between consecutive references to some $\mathrm{M}[A]$ with the longest latency for the currently simulating pre-cluster–cluster pair. In other words, the 100-th percentile of reuse latencies necessarily encompasses the maximum reuse latency.

Recall furthermore, the bijective projection of the pre-cluster–cluster period containing L instructions to the discrete interval $[1, L]$ and the partitioning of this interval into mutually-exclusive bucket_is ($i \in \{1, 2, ..., n\}$) whose union is exactly $[1, L]$. Let

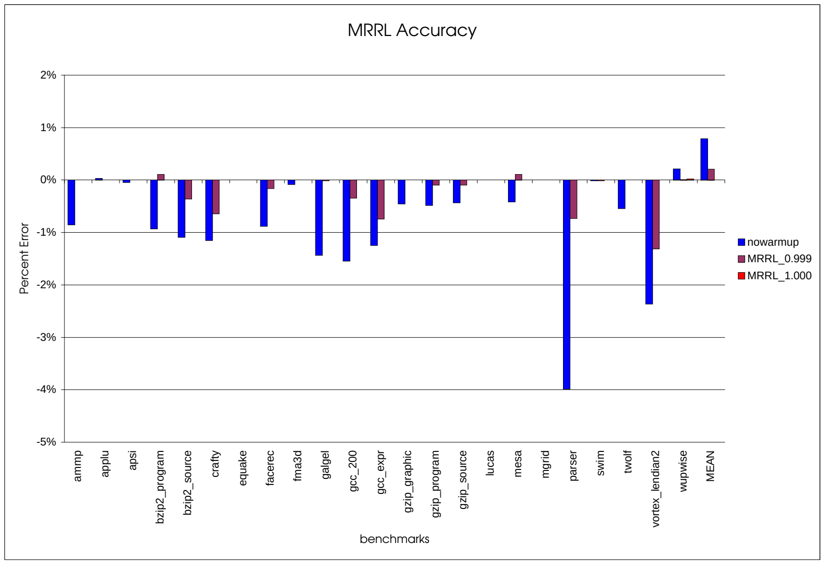

Figure 5.6: MRRL cache accuracy relative to *fullwarmup* with automatically-chosen (BBDA) samples [51]. MEAN calculated from the %-error absolute values.

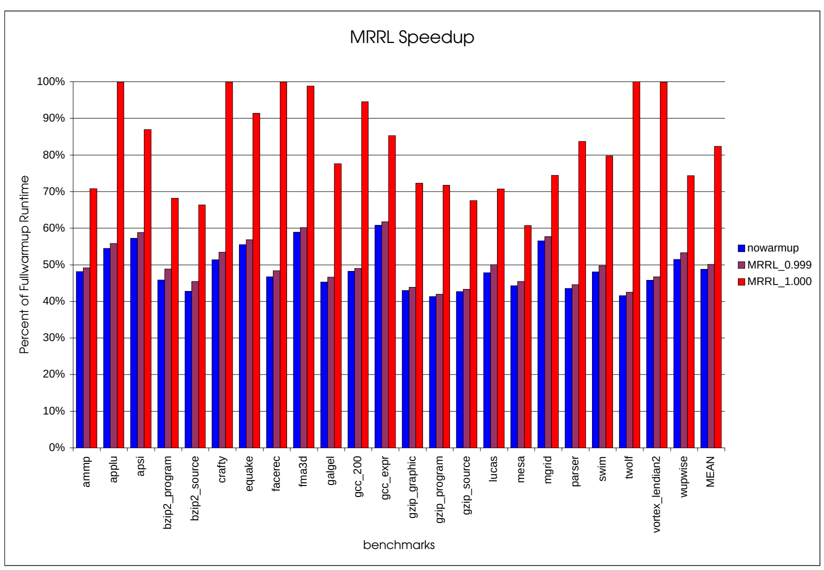

Figure 5.7: MRRL speedup relative to *fullwarmup* with automatically-chosen (BBDA) samples [51].

benchmark	*fullwarmup*	*nowarmup*	$\%_{MRRL_{0.999}}$	$\%_{MRRL_{1.000}}$
ammp	1.7985	-0.85%	0.00%	0.00%
applu	1.1744	0.03%	0.00%	0.00%
apsi	2.5419	-0.04%	0.00%	0.00%
bzip2_program	2.3517	-0.93%	0.11%	0.00%
bzip2_source	1.9159	-1.09%	-0.36%	0.00%
crafty	2.4337	-1.15%	-0.64%	0.00%
equake	0.5729	0.00%	0.00%	0.00%
facerec	1.6998	-0.88%	-0.16%	0.00%
fma3d	1.0634	-0.08%	0.00%	0.00%
galgel	1.9898	-1.43%	-0.01%	0.00%
gcc_200	1.9767	-1.54%	-0.34%	0.00%
gcc_expr	2.1749	-1.24%	-0.74%	0.00%
gzip_graphic	2.2688	-0.45%	0.00%	0.00%
gzip_program	2.3459	-0.48%	-0.09%	0.00%
gzip_source	2.1670	-0.43%	-0.09%	0.00%
lucas	0.9860	0.00%	0.00%	0.00%
mesa	2.3684	-0.41%	0.11%	0.00%
mgrid	1.1071	0.00%	0.00%	0.00%
parser	1.6241	-3.98%	-0.73%	0.00%
swim	0.7661	-0.01%	-0.01%	0.00%
twolf	1.5616	-0.54%	0.00%	0.00%
vortex_lendian2	2.3797	-2.36%	-1.31%	0.00%
wupwise	1.9799	0.21%	0.01%	0.02%
MEAN		0.85%	0.24%	0.00%

Table 5.12: MRRL accuracy using automatically-chosen (BBDA) samples [51]. MEAN calculated from the absolute value of the %-error measurements.

$w_{1.000} \mapsto$ bucket$_i$, where bucket$_i$ represents the interval subset $[a, b]$ for $a \geq 1$, $b \leq L$ and b is greater than or equal to the maximum reuse latency instruction count. My objective is to accurately produce cache and branch predictor state at the beginning of each sample cluster just as it would have appeared using *fullwarmup*. I assert that if 100% of consecutive accesses to each unique M[A] that occur during the current pre-cluster–cluster pair span $w_{1.000}$ or fewer instructions, then those references

benchmark	*fullwarmup*	*nowarmup*	$\%_{MRRL_{0.999}}$	$\%_{MRRL_{1.000}}$
ammp	136652 sec.	48.31%	97.83%	52.96%
applu	88173 sec.	54.49%	97.60%	0.00%
apsi	103645 sec.	57.26%	97.33%	48.16%
bzip2_program	41055 sec.	45.81%	93.36%	51.20%
bzip2_source	24427 sec.	42.73%	93.73%	44.65%
crafty	44730 sec.	51.41%	96.00%	0.00%
equake	57487 sec.	55.50%	97.61%	35.42%
facerec	70352 sec.	46.71%	96.45%	0.00%
fma3d	62895 sec.	58.90%	97.86%	32.19%
galgel	163568 sec.	45.25%	96.99%	28.53%
gcc_200	48385 sec.	48.20%	98.35%	3.89%
gcc_expr	5221 sec.	60.83%	98.46%	59.86%
gzip_graphic	43311 sec.	43.01%	98.04%	31.95%
gzip_program	61388 sec.	41.34%	98.42%	26.43%
gzip_source	31862 sec.	42.71%	98.64%	41.85%
lucas	48603 sec.	47.80%	95.26%	52.02%
mesa	119856 sec.	44.25%	97.22%	62.73%
mgrid	125315 sec.	56.55%	97.99%	68.36%
parser	229005 sec.	43.57%	97.75%	7.84%
swim	85202 sec.	48.05%	96.51%	34.10%
twolf	136146 sec.	41.54%	97.61%	0.00%
vortex_lendian2	44434 sec.	45.75%	97.87%	0.00%
wupwise	109109 sec.	51.46%	96.49%	55.56%
MEAN		48.68%	97.07%	33.02%

Table 5.13: MRRL achieved potential speedup on automatically-chosen (BBDA) samples [51]. Running time measured in seconds.

that occur more than $w_{1.000}$ instructions prior to the cluster will not be reaccessed during the cluster. I furthermore assert that these pre-$w_{1.000}$ references are therefore irrelevant to the cluster, and do not need to be modeled.

Hence, precisely as is demonstrated in Table 5.12, all the benchmarks except *wupwise* have $\text{MRRL}_{1.000}$ percent error IPC deviations of zero. I speculate that *wupwise*'s 0.02% deviation is very likely due to the "stale" state simulation philosophy used in these simulations. Conte *et al.* showed [8] that flushing branch predictor state at the conclusion of each sample cluster has a detrimental effect on the accuracy of measurements taken during the clusters, and advocates a stale state approach whereby cache and branch predictor state as it appeared at the conclusion of cluster $n-1$ is preserved as the basis for warm up of pre-cluster region n. Thus, what probably occurred in *wupwise* was that some small amount of cache or branch predictor state that would have been altered or evicted by *fullwarmup*, managed to survive until the next cluster, where it slightly bolstered instruction throughput.

Unfortunately, while $\mathrm{MRRL}_{1.000}$ achieves superior accuracy, it does not speed up simulation as effectively as MRRL_N for $N \in (0,1)$. Table 5.13 (and Figure 5.7) shows the achieved potential speed up for $\mathrm{MRRL}_{0.999}$ and $\mathrm{MRRL}_{1.000}$. In every case, $\mathrm{MRRL}_{0.999}$ achieves more than 90% of the maximum possible speed up (represented by *nowarmup*). $\mathrm{MRRL}_{1.000}$ on the other hand, achieves much less of the potential, with several benchmarks—*applu*, *crafty*, *facerec*, *twolf*, *vortex_lendian2*—seeing no speed up at all, and two—*gcc_200*, *parser*—which achieve very little. This minute acceleration is due to the large maximum reuse latency which is just as long as entire pre-cluster phase; in these cases, $\mathrm{MRRL}_{1.000}$ degenerates to *fullwarmup*. Nevertheless, $\mathrm{MRRL}_{1.000}$ still achieves roughly 33% of potential speed up on average which is respectable given $\mathrm{MRRL}_{1.000}$'s unimpeachable accuracy. On this final suite of experiments, the MRRL technique once again vindicates itself nicely.

Component	Configuration
L1 Instruction cache	64KB, 2-way, 32B blocks, 1-cycle access latency
L1 Data cache	64KB, 2-way, 32B blocks, 1-cycle access latency
Unified L2 cache	1024KB, 4-way, 64B blocks, 11-cycle access latency
Unified L3 cache	8192KB, 8-way, 128B blocks, 31-cycle access latency

Table 5.14: Simulator 3-level cache hierarchy configuration for MRRL experiments.

5.5 MRRL: Three-level Cache Hierarchy

As stated in Chapter 4 in the discussion of MRRL's advances over MSE, MRRL is able to accurately warm up any cache hierarchy depth regardless of unified levels or cache block widths, from a single profile. To illustrate this point, I now include MRRL speed up and accuracy measurements (both relative to *fullwarmup*) for experiments conducted with a simulated 3-level cache hierarchy (reminiscent of the IBM POWER4 [65], and Intel Itanium [48]). The exact cache configuration is detailed in Table 5.14. Both the second level and third level are unified, hosting both instructions and data. Notice the different block widths for each level of the cache (*i.e.*, primary, 32 bytes; secondary, 64 bytes; tertiary, 128 bytes). While admittedly contrived, this setup makes an effective demonstration of MRRL's indifference to block width.

Furthermore, as discussed in Chapter 4, per benchmark–input pair and sample, MRRL can make repeated use of a single profiling run; to demonstrate this point, these experiments recycle the BBDA samples and profile data from 5.4. MRRL's one-size-fits-all profiling advantage is especially beneficial because it allows researchers to amortize profiling cost over many over experiments, with many pipeline configurations. In these experiments the cache organization has been altered, but conceivably

benchmark	*fullwarmup*	*nowarmup*	$\%_{MRRL_{0.999}}$
ammp	1.9111	-1.29%	-0.05%
applu	1.4291	-0.06%	-0.01%
apsi	1.5756	0.25%	-0.01%
bzip2_program	2.4470	-1.38%	-0.23%
bzip2_source	2.2885	-1.87%	-0.25%
crafty	2.4092	-1.33%	-0.02%
equake	0.9949	-0.24%	-0.01%
facerec	2.5364	-3.32%	-0.06%
fma3d	1.3312	-0.29%	0.02%
galgel	2.1453	-1.34%	-0.17%
gcc_200	2.2865	-1.57%	-0.83%
gcc_expr	2.2831	-1.93%	-1.05%
gzip_graphic	2.2788	-0.26%	0.00%
gzip_program	2.3497	-0.33%	0.00%
gzip_source	2.1616	-0.34%	0.00%
lucas	1.1080	0.00%	0.00%
mesa	2.6466	-0.68%	-0.24%
mgrid	1.8048	-0.84%	-0.01%
parser	1.7130	-6.56%	0.00%
swim	1.0229	-0.03%	-0.01%
twolf	1.4950	-1.24%	0.00%
vortex_lendian2	2.5308	-3.83%	-0.56%
wupwise	1.8661	-0.43%	-0.01%
MEAN		1.25%	0.28%

Table 5.15: $\text{MRRL}_{0.999}$ accuracy using automatically-chosen (BBDA) samples [51] in a three-level cache. MEAN calculated from the absolute value of the %-error measurements.

benchmark	*fullwarmup*	*nowarmup*	$\%_{MRRL_{0.999}}$
ammp	98684 sec.	55.15%	97.69%
applu	70376 sec.	57.02%	97.27%
apsi	126092 sec.	58.34%	97.32%
bzip2_program	49555 sec.	50.01%	96.84%
bzip2_source	19305 sec.	47.56%	86.85%
crafty	28809 sec.	52.30%	91.22%
equake	55432 sec.	53.57%	88.32%
facerec	75632 sec.	45.65%	96.99%
fma3d	33345 sec.	58.07%	97.26%
galgel	161327 sec.	52.38%	97.06%
gcc_200	44052 sec.	55.74%	98.54%
gcc_expr	5036 sec.	68.15%	97.47%
gzip_graphic	27217 sec.	52.99%	97.13%
gzip_program	60267 sec.	47.30%	92.65%
gzip_source	27675 sec.	51.88%	97.07%
lucas	34005 sec.	48.61%	96.61%
mesa	73539 sec.	50.69%	97.20%
mgrid	144767 sec.	57.72%	96.48%
parser	210676 sec.	50.11%	95.34%
swim	81925 sec.	49.57%	93.11%
twolf	136197 sec.	42.66%	95.15%
vortex_lendian2	31729 sec.	51.63%	97.63%
wupwise	79583 sec.	44.96%	91.60%
MEAN		53.10%	97.04%

Table 5.16: $\text{MRRL}_{0.999}$ achieved potential speedup on automatically-chosen (BBDA) samples [51] in a three-level cache. Running time measured in seconds.

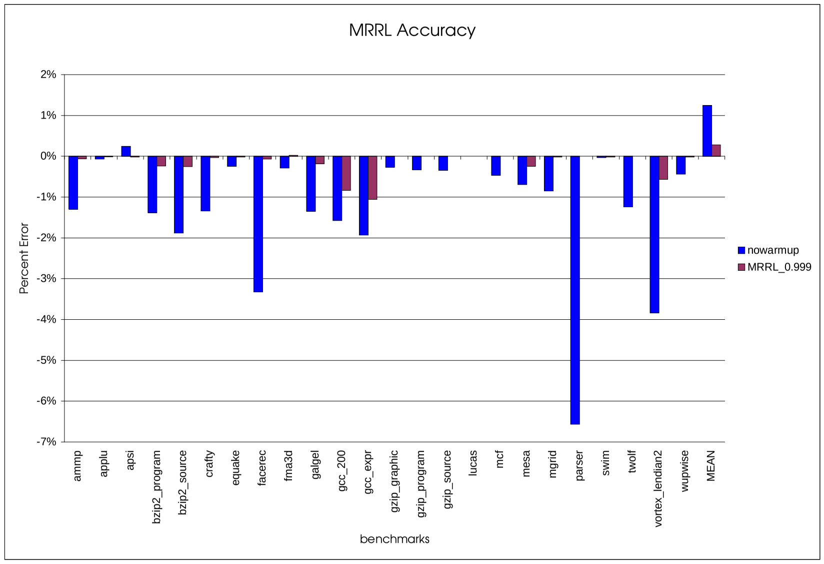

Figure 5.8: Three-level cache accuracy relative to *fullwarmup*.

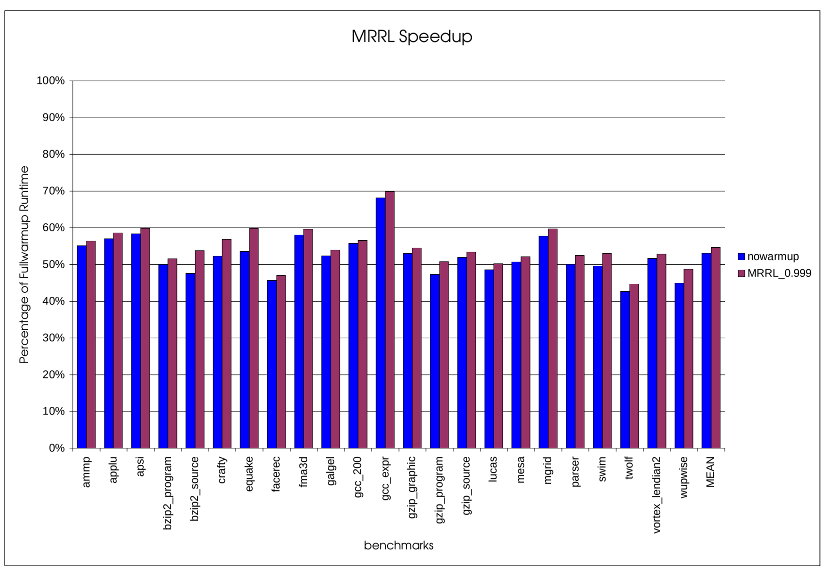

Figure 5.9: Three-level cache speedup relative to *fullwarmup*.

researchers could explore many more dimensions of the design space, including re-order buffer depth, number and type of functional units, branch predictor organization, commit bandwidth, issue capacity and algorithm, memory bandwidth, register file size and access latency, and value prediction algorithms, all from a single profile.

Since MRRL with $N = 0.999$ proved so successful in terms of accuracy [3] and speed in 5.4, other than *fullwarmup* and *nowarmup*, $\mathrm{MRRL}_{0.999}$ is the only configuration modeled in these experiments. Table 5.15 (and Figure 5.8) illustrates that $\mathrm{MRRL}_{0.999}$ maintains simulation accuracy even when used to warm up three levels of cache, never diverging by more than 1.05% (for *gcc_expr*) from the *fullwarmup* IPC. Table 5.15 also shows that while *nowarmup*'s mean percent-error (calculated from the absolute values of the benchmark deviations) is small, at 1.25%, an individual benchmark such as *parser* can diverge much more substantially. This once again demonstrates that insufficient warm up cannot be trusted to produce accurate simulation.

Table 5.16 (and Figure 5.9) shows the achieved potential speed up of each benchmark. In nearly all cases, the $\mathrm{MRRL}_{0.999}$ achieved potential speed up is greater than 90%; the only exceptions, *bzip2_source* and *equake*, still achieve greater than 80% nevertheless.

5.6 MRRL & *MSEwarmup*: Statistical Accuracy Analysis

This section assesses the accuracy of MRRL and *MSEwarmup* with respect to *fullwarmup*. To facilitate this analysis, the experiments conducted for this section use

[3] In 5.6, I show that in terms of simulation accuracy, $N = 0.999$ varies by a statistically insignificantly amount from *fullwarmup*.

random cluster sampling as described by Conte *et al.* [8] which is amenable to rigorous statistical analysis (based on the fact that all regions of the end-to-end dynamic instruction stream had uniform probability of being selected for inclusion in the sample). I exploit this amenability to quantitatively prove the null hypothesis

H_0: Instruction throughput (IPC) as measured by MRRL for $N \in \{0.990, 0.999\}$ and *MSEwarmup* for $p \in \{95.0\%, 99.9\%\}$ deviates by a statistically insignificant amount from instruction throughput as measured by *fullwarmup* at the 5% level of significance.

In other words, for $\alpha = 0.05$, H_0: $\mathrm{IPC}_{fullwarmup}$ - $\mathrm{IPC}_{MRRL_{\{0.990,0.999\}}, MSE_{\{95.0\%,99.9\%\}}} = 0$, where α is the probability of failing to reject an untrue hypothesis.

In each experiment, clusters containing 1 million contiguous instructions apiece were chosen at random from the end-to-end dynamic instruction stream of each benchmark; this is in contrast to the larger clusters used in the previous experiments. Conte *et al.* use clusters of only 100,000 instructions apiece, and Sherwood *et al.* [50, 51] use clusters with instruction counts that are integer multiples of 100 million. When choosing a cluster size, it is important to choose a size that is not too small to measure interesting behavior. A cluster size of only 1,000 instructions for example, would pose very little challenge for a modern pipeline capable of tracking one hundred or more in-flight (*i.e.*, partially-executed) instructions. My hypothesis was that smaller clusters, more in line with [8] would cumulatively estimate the true, end-to-end IPC with good accuracy. (For sufficiently large samples, the experimental data show that this hypothesis is correct; a possible avenue for future research would be to experiment with MRRL using random cluster sampling with varying cluster sizes.) The cluster size of 1 million instructions conservatively manipulated the random cluster sampling approach used in [8] since increasing the cluster size increases the amount of behavior modeled and measured per cluster.

To select the clusters, each benchmark was first simulated by *sim-fast*—the rapid instruction-level simulator from the SimpleScalar [2, 4] toolset—to obtain the end-to-end dynamic instruction count. Next, a simple Perl script was used to select the 1-million-instruction clusters at random[4] from the discrete interval $[1, L]$, where L is the dynamic instruction count. The locations (as the number of completed instructions relative to the start of execution) of the clusters were saved to a file, and subsequently used to drive the multiple cluster profiling and simulation steps enumerated in Chapter 3 and Chapter 4. For each benchmark, the same set of sample clusters was used to experiment with all four warm up techniques (*fullwarmup*, MRRL, *MSEwarmup*, and *nowarmup*).

Sampling (whether random, systematic, stratified, cluster, or multistage [22]) always produces error because only a subset of a population is measured rather than the entire population. Hence, by sampling, one can only *estimate* the characteristics of an entire population. Random cluster sampling allows one to rigorously gauge the amount of error and the probability that the amount is significant, based upon the assumption that all members of the population had uniform probability of being

[4] By "at random," I mean such that all regions of the discrete interval $[1, L]$ have equal probability of being selected.

included in the sample. Increasing the size of a sample increases the accuracy of the estimation by drawing the estimation value asymptotically nearer to the true value[5]. A key consideration therefore, was to determine the number of clusters to draw from each benchmark. For most benchmarks, 50 clusters were sufficient to estimate[6] the end-to-end IPC (*i.e.*, IPC_{true}) when simulated with *fullwarmup*. For *applu* and *galgel*, however, a larger sample had to be drawn to obtain good accuracy; for these, I used samples of 500 1-million-instruction clusters.

For the MRRL simulations, the warm phase was engaged w_N instructions prior to each cluster for $N \in \{0.990, 0.999\}$. $N = 0.999$ has shown consistently good performance in accurately mimicking the performance of *fullwarmup*. $N = 0.990$ was chosen to test whether the same performance could be demonstrated for a lower value of N. If a lower value for N performs as well as $N = 0.999$ (*i.e.*, also deviates from *fullwarmup* by a statistically insignificant amount), then this lower value of N establishes a tighter lower bound on the minimal necessary N to achieve accurate simulation. If not, then the threshold minimal N can be said to exist somewhere in the interval $(0.990, 0.999]$. As will be shown in 5.6.3, the former scenario is true and therefore establishes a tighter threshold N. I do not experiment with lower values for N since, as was demonstrated in 5.3.1, $\mathrm{MRRL}_{0.990}$ achieves over 97% of the maximum potential speedup on average and deviated from the *fullwarmup* IPC by only 2.42% in the worst case.

[5] By "true value" I refer to the value that would be obtained by measuring the entire population.

[6] My threshold for a good estimate was to deviate from the end-to-end IPC by less than 5% when the sample is simulated with *fullwarmup*.

Cache Hierarchy	
L1 Data	16KB; 4-way assoc., 32B lines, 2-cycle hit
L1 Instruction	8KB; 2-way assoc., 32B lines, 2-cycle hit
L2 Unified	1MB; 4-way assoc., 64B lines, 20-cycle hit
Combined Branch Predictor	
Bimodal	8192 entries
PAg	8192 entries
Return Address Stack	64 entries
Branch Target Buffer	2048 entries; 4-way assoc.

Table 5.17: Configuration of simulated cache and branch predictor.

For the *MSEwarmup* experiments, I tested $p \in \{95.0\%,99.9\%\}$ as the probabilities of accurate warm up. The other MSE parameters were dictated by the first-level cache, shown in the cache and branch predictor configuration Table 5.17. While the block width of both the L1 data cache and the L1 instruction cache are identical (32 bytes), the data cache has twice the capacity of the instruction cache. Since the data cache is larger, its dimensions guided the MSE calculation thus: $m = \text{MSE}(128,4,p)$. Hence, warm up for the *MSEwarmup* experiments was driven entirely by the first level of cache. For the reasons cited in Chapter 4, *MSEwarmup* is not well suited to warming up large structures or branch predictors. (Branch prediction defaulted to *fullwarmup* for these experiments, thereby assuring sound warm up of the branch prediction buffer.) Furthermore, the MRRL warm up intervals—which have been shown to accurately initialize all levels of cache—are usually small, spanning only a fraction of the lattermost pre-cluster instruction stream. If the warm up intervals generated by *MSEwarmup* for the first-level caches are larger than those generated by MRRL, then the simulation accuracy will not be compromised; unfortunately, neither will the simulation running time decrease.

Once each benchmark's sample was selected, the next step was to profile to gather MSE/MRRL data for each benchmark. A Perl script was then used to extract the *MSEwarmup* t and MRRL w_N for each benchmark's pre-cluster–cluster pairs. When fed to the multiple cluster simulator, these data were used to demarcate the boundary between the pre-cluster cold phase and warm phase. The previously chosen hot phases (clusters) remained fixed just as they were during the profile.

The metrics used to measure *MSEwarmup*'s and MRRL's merit are percent-error IPC deviation from *fullwarmup*, accuracy with respect to the true IPC, statistical significance of deviation by matched-pairs t-test, and running time as a percentage of *fullwarmup*. For completeness and as a basis the concluding discussion of simulation acceleration, I also include data arising from *nowarmup* for each of the metrics aforementioned. (Recall that *nowarmup* merely recycles state from one cluster to the next, and models caching and branch prediction solely during sample clusters.)

5.6.1 IPC Accuracy compared to IPC$_{fullwarmup}$

For each benchmark, Table 5.18 shows the true end-to-end IPC[7] (*i.e.*, IPC$_{true}$) generated by simulating in cycle-accurate detail for the entire dynamic instruction stream, *fullwarmup* IPC (*i.e.*, IPC$_{fullwarmup}$) percent-error deviation relative to IPC$_{true}$, and *nowarmup* IPC (*i.e.*, IPC$_{nowarmup}$) percent-error deviation relative to IPC$_{fullwarmup}$.

[7] Most of these IPCs come from the *SimPoint* [49] Web site. They were generated for a specific configuration of *sim-outorder* (linked to from the site). *MSEwarmup*, MRRL, *fullwarmup*, and *nowarmup* experiments compared against these IPCs use the same *sim-outorder* configuration and the same benchmark binaries.

benchmark	IPC_{true}	$\mathrm{IPC}_{fullwarmup}$	$\mathrm{IPC}_{nowarmup}$
applu	0.831	-0.36%	-1.09%
apsi	1.008	3.12%	-2.23%
art_110	0.598	-0.57%	0.34%
crafty	0.569	-3.64%	-0.80%
equake	0.310	0.42%	2.22%
facerec	1.446	-4.87%	-10.46%
fma3d	0.535	-0.37%	1.57%
galgel	1.334	-0.60%	-11.61%
gzip_graphic	1.365	-3.28%	-0.52%
lucas	0.774	2.25%	0.23%
mcf	0.092	3.04%	0.84%
mgrid	0.987	4.72%	-1.87%
twolf	0.636	-1.08%	-1.76%
vortex_lendian2	1.057	-3.18%	-0.63%
vpr_route	1.023	0.18%	-1.16%
MEAN		2.11%	2.49%

Table 5.18: $\mathrm{IPC}_{fullwarmup}$ %-error relative to IPC_{true} ($100\% \cdot \frac{IPC_{fullwarmup}-IPC_{true}}{IPC_{true}}$); and $\mathrm{IPC}_{nowarmup}$ %-error relative to $\mathrm{IPC}_{fullwarmup}$ ($100\% \cdot \frac{IPC_{nowarmup}-IPC_{fullwarmup}}{IPC_{fullwarmup}}$). MEAN calculations based on the absolute value of errors.

In other words, Table 5.18 compares the sample mean for $\mathrm{IPC}_{nowarmup}$ to the sample mean for $\mathrm{IPC}_{fullwarmup}$, which is in turn compared to the end-to-end IPC_{true}. Notice the *nowarmup* percent-error deviation from $\mathrm{IPC}_{fullwarmup}$ for the benchmarks *facerec* and *galgel* of -10.46% and -11.61%, respectively. These deviations—substantially larger than those of the other benchmarks—are evidence that inadequate warm up can compromise simulation accuracy. More rigorous, quantitative evidence of the unreliability of inadequate warm up will be given in 5.6.2 and 5.6.3.

Before developing a more formal framework for *MSEwarmup* and MRRL accuracy analysis, it is important to define the components of error and their application to microprocessor simulation. Henry [22] separates error into two components: sampling and nonsampling. Sampling error is an unavoidable consequence of the fact that a sample can only approximately capture characteristics of an entire population. Nonsampling error arises from a failure to ensure the representativeness of the environment in which the sample measurements are taken. In other words, if the environment of the sample does not at least approximate the environment of the population, the measurements taken by sampling will tend to be skewed. In a microprocessor pipeline, the state of the cache and the branch predictor heavily influence instruction throughput [21] and their state constitutes the instruction stream execution environment. Failure to accurately initialize state within the simulated cache and branch predictor may adversely affect measurements taken during cycle-accurate simulation of the sample clusters; this is the cold-start effect. By modeling all pre-cluster cache and branch predictor interactions however, *fullwarmup* simulation is impervious to nonsampling error; hence, my research objective can also be considered to develop a warm up strategy that accelerates warm up without adding additional nonsampling error (i.e., a warm up strategy that yields results that are identical or closely approximate results generated by *fullwarmup*) [8].

Using Henry's error component terminology, it can be stated that Table 5.18 qualitatively demonstrates that inadequate warm up ($\text{IPC}_{nowarmup}$) generated substantial additional nonsampling error for *facerec* and *galgel*. Table 5.19 on the other hand, indicates that *MSEwarmup* and MRRL do not generate a large amount of nonsampling error. Table 5.19 lists the *fullwarmup* IPCs, and percent-error deviations therefrom

		%-error			
benchmark	$\text{IPC}_{fullwarmup}$	$\text{MRRL}_{0.999}$	$\text{MRRL}_{0.990}$	$\text{MSE}_{99.9\%}$	$\text{MSE}_{95.0\%}$
applu	0.828	0.01%	0.29%	0.00%	0.00%
apsi	1.039	-0.01%	-0.04%	0.03%	0.03%
art_110	0.595	0.00%	0.00%	0.00%	0.00%
crafty	0.548	-0.02%	-0.04%	0.09%	0.09%
equake	0.311	0.00%	0.00%	0.00%	0.00%
facerec	1.376	0.18%	0.36%	0.63%	0.98%
fma3d	0.533	3.90%	3.88%	3.94%	3.94%
galgel	1.326	-0.20%	-0.62%	0.02%	0.02%
gzip_graphic	1.320	-0.09%	-0.01%	0.01%	0.01%
lucas	0.791	-0.04%	-0.13%	-0.28%	-0.28%
mcf	0.095	0.00%	0.00%	0.00%	0.00%
mgrid	1.034	-0.01%	-0.01%	-0.01%	-0.09%
twolf	0.629	0.13%	0.14%	0.19%	0.19%
vortex_lendian2	1.023	0.06%	0.07%	0.31%	0.13%
vpr_route	1.025	0.00%	0.00%	0.00%	0.00%
MEAN		0.31%	0.37%	0.05%	0.33%

Table 5.19: IPC %-error relative to $\text{IPC}_{fullwarmup}$ ($100\% \cdot \frac{IPC_X - IPC_{fullwarmup}}{IPC_{fullwarmup}}$). MEAN calculations based on the absolute value of errors.

for MRRL at $N = 0.999$ and $N = 0.990$ (*i.e.*, $\text{MRRL}_{0.999}$ and $\text{MRRL}_{0.990}$), and *MSE-warmup* at $p = 99.9\%$ and $p = 95.0\%$ (*i.e.*, $\text{IPC}_{MSE_{99.9\%}}$ and $\text{IPC}_{MSE_{95.0\%}}$). For all benchmarks except *fma3d*, the percent difference deviation from *fullwarmup* is less than 0.7%. *fma3d*'s seemingly drastic nonconformance however, is due to the small numbers involved in the percent-error calculation. Take for example the largest deviation of 3.94%, due to *MSEwarmup* at $p = 95.0\%$ and $p = 99.9\%$; $\text{IPC}_{fullwarmup}$ = 0.533, $\text{IPC}_{MSE_{95.0\%,99.9\%}} = 0.554$. The relative error, $100\% \cdot (\frac{0.554-0.533}{0.533}) = 3.94\%$ makes the deviation look much worse than it really is when one considers that the absolute error is so small: 0.554 - 0.533 = 0.021, or 21 thousands of an instruction

benchmark	IPC_{true}	$IPC_{MRRL_{0.999}}$	$IPC_{MRRL_{0.990}}$	$IPC_{MSE_{99.9\%}}$	$IPC_{MSE_{95.0\%}}$
applu	0.831	0.829±0.053	0.831±0.053	0.828±0.053	0.828±0.053
apsi	1.008	1.039±0.063	1.039±0.064	1.040±0.064	1.040±0.064
art_110	0.597	0.595±0.029	0.595±0.029	0.595±0.029	0.595±0.029
crafty	0.569	**0.548±0.014**	**0.548±0.014**	**0.549±0.014**	**0.549±0.014**
equake	0.310	0.311±0.104	0.311±0.104	0.311±0.104	0.311±0.104
facerec	1.446	1.378±0.460	1.381±0.460	1.384±0.458	1.389±0.458
fma3d	0.535	0.554±0.058	0.554±0.058	0.554±0.061	0.554±0.058
galgel	1.334	1.323±0.112	1.317±0.112	1.326±0.112	1.326±0.112
gzip_graphic	1.365	1.319±0.094	1.320±0.094	1.320±0.094	1.320±0.094
lucas	0.774	0.791±0.157	0.790±0.156	0.789±0.157	0.789±0.157
mcf	0.092	0.095±0.052	0.095±0.052	0.095±0.052	0.095±0.052
mgrid	0.987	1.034±0.106	1.034±0.106	1.034±0.106	1.033±0.106
twolf	0.636	**0.629±0.004**	**0.630±0.004**	**0.630±0.004**	**0.630±0.004**
vortex_lendian2	1.057	1.024±0.040	1.024±0.040	1.027±0.040	1.025±0.040
vpr_route	1.023	1.025±0.038	1.025±0.038	1.025±0.038	1.025±0.038

Table 5.20: IPC 95% confidence intervals centered around $\overline{IPC}$ (the overall sample IPC), for $MRRL_{0.999}$, $MRRL_{0.990}$, $MSEwarmup_{99.9\%}$, and $MSEwarmup_{95.0\%}$. Bold entries fail to accurately predict the amount of sampling error.

per cycle.

5.6.2 IPC Accuracy compared to IPC_{true}

While $MRRL_{0.999}$, $MRRL_{0.990}$, $MSEwarmup_{99.9\%}$ and $MSEwarmup_{95.0\%}$ are apparently sound warm up strategies, and *nowarmup* apparently unsound, I will now rigorously demonstrate these hypotheses. For each benchmark, the mean instruction throughput was measured by counting the number of cycles consumed in executing the sample clusters. Dividing the total number of executed instructions by this amount yielded the overall sample IPC (*i.e.*, $\overline{IPC}$). For a well-chosen sample, this sample IPC will be a good estimate of the end-to-end IPC. The *standard error* is a useful tool to analyze the goodness of a sample estimate [13, 59]. The standard error is computed as the

benchmark	IPC_{true}	$\text{IPC}_{fullwarmup}$	$\text{IPC}_{nowarmup}$
applu	0.831	0.828±0.053	0.819±0.053
apsi	1.008	1.039±0.063	1.039±0.064
art_110	0.597	0.595±0.029	0.597±0.029
crafty	0.569	**0.548±0.014**	**0.544±0.014**
equake	0.310	0.311±0.104	0.318±0.110
facerec	1.446	1.376±0.460	**1.232±0.135**
fma3d	0.535	0.533±0.061	0.542±0.055
galgel	1.334	1.326±0.112	**1.172±0.104**
gzip_graphic	1.365	1.320±0.094	1.313±0.094
lucas	0.774	0.791±0.157	0.793±0.144
mcf	0.092	0.095±0.052	0.096±0.050
mgrid	0.987	1.034±0.106	1.014±0.080
twolf	0.636	**0.629±0.004**	**0.618±0.009**
vortex_lendian2	1.057	1.023±0.040	1.017±0.040
vpr_route	1.023	1.025±0.038	1.013±0.036

Table 5.21: 95% IPC confidence intervals centered around $\overline{IPC}$ (the overall sample IPC), for *fullwarmup*, and *nowarmup*. Successful simulations contain IPC_{true} within their confidence interval. Bold entries fail to predict the amount of sampling error.

quotient of the per-cluster sample standard deviation in IPC and the square root of the number of clusters:

$$s_{\overline{IPC}} = \frac{\sigma}{\sqrt{\#_{cluster}}}$$

I assume (as in [8]) that error is normally distributed[8]; hence, the 95% confidence interval is $\overline{IPC}\pm 1.96 s_{\overline{IPC}}$. In other words, for a well-chosen sample, one can assume $\text{IPC}_{true} \in [\overline{IPC} - s_{\overline{IPC}}, \overline{IPC} + s_{\overline{IPC}}]$ with 95% certainty.

Furthermore, let $e = \text{IPC}_{true} - \overline{IPC}$; if $[\text{IPC}_{true} - e, \text{IPC}_{true} + e] \subset [\overline{IPC} - 1.96 s_{\overline{IPC}},$

[8] The assumption of normality is safe since the samples contain 50 clusters apiece. Samples of 30 or fewer elements would use the Student's-t distribution [59] with $\#_{cluster} - 1$ degrees of freedom.

$\overline{IPC}+1.96s_{\overline{IPC}}$], then the relative error between IPC_{true} and $\overline{IPC}$ was accurately predicted by the 95% confidence interval. Table 5.20 shows that the relative error between $\mathrm{IPC}_{MRRL_{0.999}}$ and IPC_{true}, $\mathrm{IPC}_{MRRL_{0.990}}$ and IPC_{true}, $\mathrm{IPC}_{MSE_{99.9\%}}$ and IPC_{true}, and $\mathrm{IPC}_{MSE_{95.0\%}}$ and IPC_{true} was predicted by every benchmark's respective 95% confidence intervals except for *crafty*, and *twolf* (in bold typeface). Table 5.21 shows however, that the 95% confidence interval failed to predict the relative error between $\mathrm{IPC}_{fullwarmup}$ and IPC_{true} for these same two benchmarks! Since *fullwarmup* perfectly models all pre-cluster cache and branch predictor interactions, it impervious to non-sampling error; hence, its failure to predict the relative error for these benchmarks is attributable to sampling error. Perfectly mimicking *fullwarmup* in this way is further evidence that MRRL at $N = 0.999$ and $N = 0.990$, and *MSEwarmup* at $p = 99.9\%$ and $p = 95.0\%$ do well at approximating *fullwarmup*. In other words, $\mathrm{MRRL}_{0.999}$, $\mathrm{MRRL}_{0.990}$, $MSEwarmup_{99.9\%}$, and $MSEwarmup_{95.0\%}$ do well at eliminating nonsampling error.

In contrast, consider the $\mathrm{IPC}_{nowarmup}$ sample means of *facerec* and *galgel*. The *nowarmup* result does not successfully predict their relative error deviation from IPC_{true}. This evidence rigorously and quantitatively confirms the hypothesis that their respective -10.46% and -11.61% percent-error deviations from the $\mathrm{IPC}_{fullwarmup}$ sample means are statistically significant.

5.6.3 IPC Accuracy according to Matched-Pairs t-test

Statistical hypothesis testing can also be used to demonstrate the statistical significance of the difference between $\mathrm{IPC}_{fullwarmup}$ and the IPC generated by another warm up technique. In particular, I used the matched-pairs t-test to compare each benchmark's *fullwarmup* per-cluster IPCs against the per-cluster IPCs generated by the other warm up techniques. In this test, the IPC of the i-th *fullwarmup* cluster is paired with its counterpart i-th, MRRL_N or $\mathit{MSEwarmup}_p$ cluster IPC. From this set of pairs, the set of cluster IPC differences is calculated and used to compute a t-score based on the difference of the means, the standard error of the means, and their Pearson product-moment correlation coefficient [70]. If for example, one wishes to compute a t-score for the matched-pairs difference between *fullwarmup* and MRRL_N, t is computed, thus:

$$t = \frac{\mu_X - \mu_Y}{\sqrt{\sigma_X^2 + \sigma_Y^2 - 2r_{XY}\sigma_X\sigma_Y}}$$

where $\mu_X - \mu_Y$ is the difference of the *fullwarmup* and MRRL_N cluster means, σ_X and σ_Y are the standard errors among the *fullwarmup* and MRRL_N cluster IPCs[9], and r_{XY} is the Pearson product-moment correlation coefficient between the *fullwarmup* and MRRL_N cluster IPCs. This t-test takes into account that I am measuring the effects of each warm up strategy as different "treatments" of the same sample population [70]. This process was used to compare *fullwarmup* to $\mathrm{MRRL}_{0.999}$, $\mathrm{MRRL}_{0.990}$, $\mathit{MSEwarmup}_{99.9\%}$, $\mathit{MSEwarmup}_{95.0\%}$, and *nowarmup*.

At the 5% level of significance, the critical t-score[10] for 50-cluster samples is 2.0096,

[9]For the matched pairs t-test, μ_X and μ_Y are computed differently from the sample IPC ($\overline{IPC}$) mentioned in 5.6.2; rather, they are computed as the mean of per-cluster IPCs. σ_X and σ_Y are computed using μ_X and μ_Y, respectively, and are therefore also different from the σ used in 5.6.2.

[10]According to the Student's-t distribution for 49 degrees of freedom and 499 degrees of freedom.

	t-score				
benchmark	$\mathrm{MRRL}_{0.999}$	$\mathrm{MRRL}_{0.990}$	*MSEwarmup*$_{99.9\%}$	*MSEwarmup*$_{95.0\%}$	*nowarmup*
applu	0.9120	1.4163	1.5872	0.8682	**5.1690**
apsi	0.9056	0.6891	0.7492	0.7524	**2.8466**
art_110	1.0474	0.0000	2.8243	2.8243	**4.1805**
crafty	1.3793	**2.2331**	**12.249**	**12.249**	**4.5735**
equake	1.5955	0.6286	1.1372	0.5753	1.3638
facerec	1.3834	1.7334	1.8663	1.7385	**4.0786**
fma3d	1.2416	1.2242	1.2791	1.2783	0.7473
galgel	0.5862	1.1256	1.9098	1.8489	**15.593**
gzip_graphic	1.9597	1.3249	1.5652	1.5652	**2.0718**
lucas	1.1194	0.9971	0.1789	0.1771	0.6455
mcf	0.5961	1.1372	1.8005	1.3845	1.4620
mgrid	1.3420	1.2976	1.1157	0.4260	1.7249
twolf	**4.3945**	**6.4937**	**8.9313**	**8.9313**	**2.1865**
vortex_lendian2	**8.4219**	**8.9749**	**24.500**	**6.4753**	**3.3422**
vpr_route	0.9410	0.9410	0.9410	0.9410	**9.1317**

Table 5.22: Matched-pairs t-test t-scores measuring the statistical significance of cluster differences between *fullwarmup* and, $\mathrm{MRRL}_{0.999}$, $\mathrm{MRRL}_{0.990}$, *MSEwarmup*$_{99.9\%}$, *MSEwarmup*$_{95.0\%}$ and *nowarmup*. At the 5% level of significance, the critical t-score for 50-cluster samples is 2.0096, and the critical t-score for 500-cluster samples (*applu* and *galgel*) is 1.9647. Entries in bold typeface exceed the critical t-score.

and the critical t-score for the 500-cluster samples (*applu* and *galgel*) is 1.9647. In other words, any 50-cluster experiment that yields a t-score greater than 2.0096 and any 500-cluster experiment that yields a t-score greater than 1.9647 violates the null hypothesis that the tested warm up technique varies by a statistically insignificant amount from *fullwarmup*. Table 5.22 lists the t-scores of the benchmarks calculated by pairing the cluster IPCs of $\mathrm{MRRL}_{0.999}$, $\mathrm{MRRL}_{0.990}$, *MSEwarmup*$_{99.9\%}$, *MSEwarmup*$_{95.0\%}$, and *nowarmup* with the *fullwarmup* cluster IPCs. *nowarmup* violates the null hypothesis for 10 benchmarks (*applu*, *apsi*, *art_110*, *crafty*, *facerec*, *galgel*, *gzip_graphic*, *twolf*, *vortex_lendian2*, and *vpr_route*). Since the preponderance of experiments violate the null hypothesis, I conclude that the alternative hypothesis is instead true of *nowarmup*: in general, *nowarmup* does not defeat nonsampling error, causing its results to diverge by a statistically significant amount from *fullwarmup*. This, combined with the large (and statistically proven significant) deviations from *fullwarmup* for the benchmarks *facerec* and *galgel* are firm evidence that adequate warm up is essential if sampled simulation results are to be trustworthy.

Recall *fma3d*'s relatively large percent-error deviation from *fullwarmup* for all of $\mathrm{MRRL}_{0.999}$, $\mathrm{MRRL}_{0.990}$, $MSEwarmup_{99.9\%}$ and $MSEwarmup_{95.0\%}$ (5.6.1, Table 5.19). I qualitatively drew the conclusion that although the percent-error deviation was 3.9%, that because the absolute error was so small—0.021 instructions per cycle—the deviation was insignificant. Table 5.22 quantitatively confirms this conclusion since the *fma3d* t-scores are less than the critical t-score for $\mathrm{MRRL}_{0.999}$, $\mathrm{MRRL}_{0.990}$, $MSEwarmup_{99.9\%}$ and $MSEwarmup_{95.0\%}$. In other words, the differences between the IPCs generated by these techniques and *fullwarmup* are statistically insignificant at the 5% level. $\mathrm{MRRL}_{0.999}$ and $\mathrm{MRRL}_{0.990}$ violate the null hypothesis for *twolf*, and *vortex_lendian2*; and $\mathrm{MRRL}_{0.990}$ additionally violates the null hypothesis for *crafty*. While initially alarming, further inspection reveals that although their t-scores imply statistically significant deviation at the 5% level, the absolute differences ($\mathrm{IPC}_{fullwarmup}$ - $\mathrm{IPC}_{MRRL_{\{0.990,0.999\}}}$) for *twolf* and *vortex_lendian2* are all 0.001. That is, the $\mathrm{MRRL}_{0.999}$ and $\mathrm{MRRL}_{0.990}$ estimates of the *fullwarmup* IPC for the benchmarks *twolf* and *vortex_lendian2* are off by 1 one-thousandth of an instruction per cycle—an amount that I safely, albeit qualitatively assume to be insignificant.

		Speedup			
benchmark	$t_{fullwarmup}$	$\text{MRRL}_{0.999}$	$\text{MRRL}_{0.990}$	$MSEwarmup_{99.9\%}$	$MSEwarmup_{95.0\%}$
applu	80887 sec.	64.30%	63.79%	100.0%	100.0%
apsi	120925 sec.	59.56%	59.46%	97.19%	96.88%
art_110	19613 sec.	36.94%	37.22%	98.82%	98.91%
crafty	78906 sec.	48.20%	48.71%	87.01%	86.24%
equake	54675 sec.	57.11%	57.80%	91.25%	91.87%
facerec	70587 sec.	51.98%	51.16%	95.40%	95.64%
fma3d	96462 sec.	61.38%	61.25%	96.66%	96.47%
galgel	162606 sec.	55.45%	55.70%	95.69%	96.27%
gzip_graphic	26643 sec.	34.58%	34.89%	97.81%	97.39%
lucas	46730 sec.	50.67%	50.79%	100.0%	99.21%
mcf	36014 sec.	46.45%	46.79%	89.09%	89.17%
mgrid	142334 sec.	60.50%	60.41%	100.0%	100.0%
twolf	133069 sec.	44.04%	44.35%	98.91%	98.82%
vortex_lendian2	64839 sec.	41.89%	38.06%	96.72%	98.78%
vpr_route	28358 sec.	44.08%	39.75%	98.63%	98.27%

Table 5.23: Random cluster sampling acceleration relative to *fullwarmup* as a percentage: $100\% \cdot \frac{t}{t_{fullwarmup}}$.

The absolute error between $\text{MRRL}_{0.990}$ and *fullwarmup* for *crafty* is (to three decimal places) 0.000—again, a qualitatively insignificant amount.

While the raw statistical evidence provided by the t-tests implies that $N = 0.990$ is less reliable than $N = 0.999$ (by exceeding the critical t-score for 1 extra benchmark), putting these results into perspective, vis-á-vis their absolute deviation from the *fullwarmup* IPC, shows that $N = 0.990$ performs just as well as $N = 0.999$ at eliminating nonsampling error. Coupled with the over 97% achievement of maximum potential speedup demonstrated in 5.3.1, this makes $\text{MRRL}_{0.990}$ the optimal warm up technique. High accuracy is also true of the *MSEwarmup* experiments, but many of these experiments achieved a negligible acceleration over *fullwarmup*. The scarcity of unique references made it difficult for *MSEwarmup* to model the MSE-prescribed m uniques during the pre-cluster regions. Thus, for pre-cluster warm up, *MSEwarmup* degenerated (or nearly so) to *fullwarmup* and completely traded away speed for accuracy. Table 5.23 shows the speed up results relative to *fullwarmup* for the random cluster sampling experiments.

Chapter 6

Tools

Integral to this research were the tools that were developed during the course of its implementation:

1. MSE formula calculation software
2. *sim-mrrlprofile*: MSE/MRRL multiple cluster profiling software
3. *sim-inorder*: MSE/MRRL-enabled multiple cluster sampling in-order issue simulation engine
4. *sim-outorder_mrrl*: MSE/MRRL-enabled multiple cluster sampling out-of-order issue simulation engine

The MSE formula calculation software was developed independently; the profiler and the simulation engines were developed through extensive modifications to existing SimpleScalar [2] software. Specifically, *sim-mrrlprofile* was built from the instruction-level *sim-safe* simulator, *sim-inorder* was built from the cache hierarchy simulator *sim-cache*, and *sim-outorder_mrrl* was built from *sim-outorder*. Both the profiler and the MRRL-enabled out-of-order simulation engine have been refined, tested, and made available for public download from a page linked to from the Laboratory for Computer Architecture at Virginia (LAVA) Web site at *http://lava.cs.virginia.edu/*. (Figure 6.1 is a screen shot of the MRRL Web site.)

6.1 MRRL Profiler: sim-mrrlprofile

To measure the reuse latencies of individual reference addresses, *sim-mrrlprofile* used three associative arrays: one for instruction addresses, one for data addresses, one for branch addresses[1]. Each element of each associative array is an ordered pair, $[A,\#_{M[A]}]$ where $\#_{M[A]}$ is the count instructions that had completed when M[A] was last referenced. As the profiler executes each instruction it performs an associative lookup, using the memory address A (*i.e.*, the current program counter value, PC) as its key. If a corresponding array element is found, the quantity $\delta insn$ is calculated as the difference between the current count of completed instructions and $\#_{M[A]}$. *Thus, $\delta insn$ is a reuse latency measurement for M[A].* The use of $\delta insn$ will be discussed shortly. If no such corresponding element is found, an unused element in the associative array is inaugurated with PC as its key and $\#_{M[A]} = 0$. An analogous process takes place inside the associative array of branches when it is determined that the fetched instruction is a branch, as well as within the associative array of data references when it is determined that the fetched instruction is a load or store.

[1] Instruction addresses and branch addresses refer to the program counter value (PC) at the time the instruction (branch or otherwise) is fetched from the simulated main memory. Data addresses, on the other hand, refer to the address loaded/stored from/to simulated main memory.

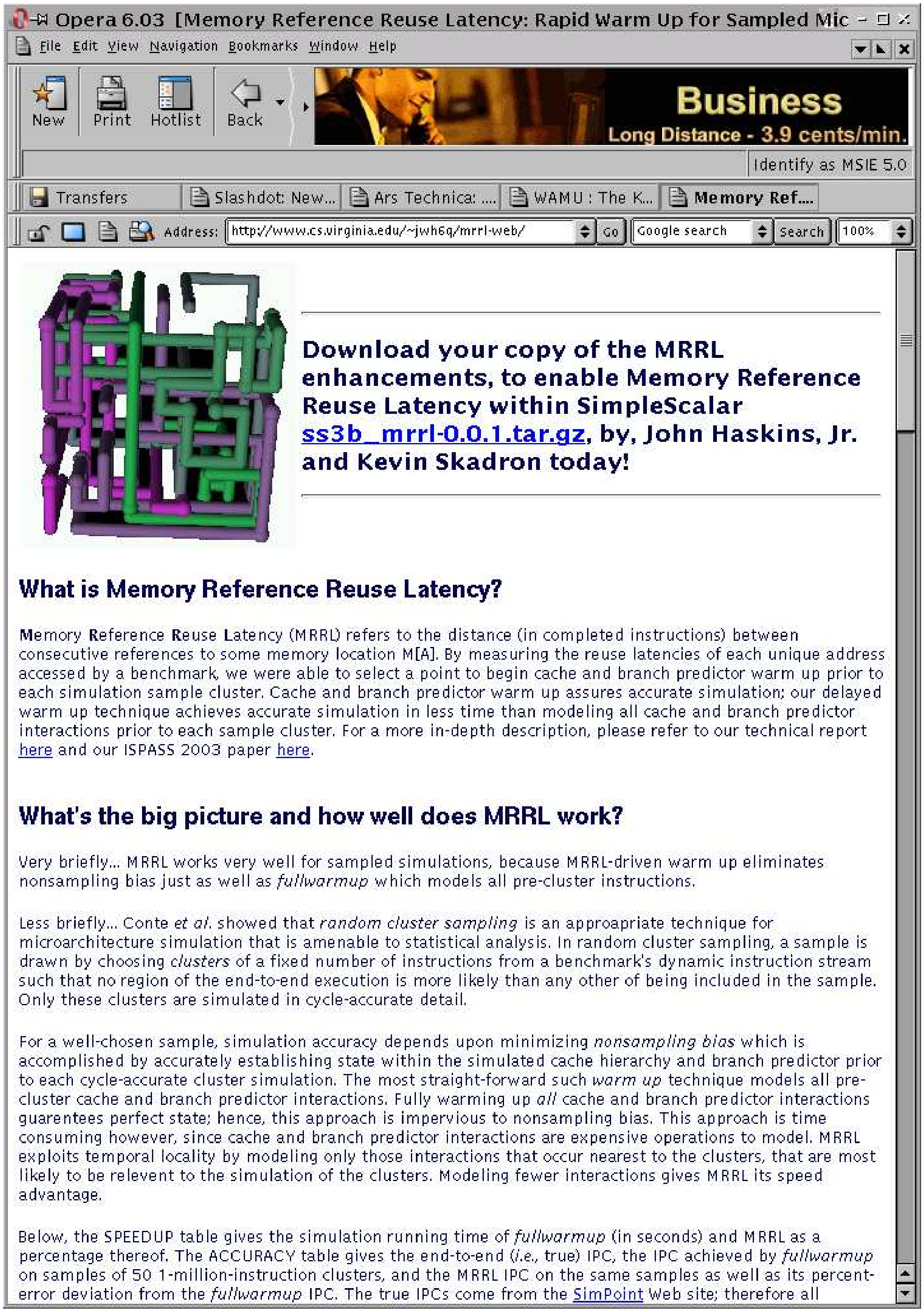

Figure 6.1: Screen shot of the MRRL Web site.

Next, because I needed to be able to group reuse latency measurements by percentile, a histogram array was also added for each reference stream (instructions, data, branches). Within a histogram array, each element corresponds to a mutually exclusive subset of the pre-cluster–cluster instructions currently being profiled. (Recall from earlier discussion, that a sample of clusters is first drawn from a benchmark's end-to-end execution; all instructions preceding each cluster, but following the previous cluster constitute pre-cluster instructions. Together, these are a pre-cluster–cluster pair.) In other words, let the currently simulating pre-cluster–cluster pair consist of instructions a through b of the total end-to-end dynamic instruction stream, and let these instructions bijectively map to the discrete closed interval $[1, b-a+1]$ thus, $\text{instruction}_k \mapsto k - a + 1$. If $[1, b-a+1]$ is partitioned into $n+1$ subsets

$$[1, k_1], [k_1 + 1, k_2], [k_2 + 1, k_3], ..., [k_n + 1, b - a + 1],$$

then each histogram array element can be mapped to one of these mutually exclusive subset partitions. For instance, according to the $[1, b-a+1]$ partition shown above the third histogram array element would hold a count of the number of reuse latency measurements that spanned as few as $k_2 + 1$ and as many as k_3 completed instructions between consecutive accesses. Whichever subset partition contains the value of the reuse latency measurement $\delta insn$ has its corresponding histogram array element incremented. Hence, if $\delta insn$ is the reuse latency measurement of a branch and $k_2 + 1 \leq \delta insn \leq k_3$, then the third element of the branch histogram array is incremented. When the profile of the current pre-cluster–cluster pair completes, the histogram array will contain a complete description of the reuse latency history of its instructions, branches, and data references.

To make *MSEwarmup* and MRRL applicable to any sampling strategy, the profiler was constructed to allow any arbitrary partitioning of benchmarks into pre-cluster–cluster sections. The alternative was to rely on MSE/MRRLprofile data gathered from each benchmark's end-to-end execution as a whole. I rejected this alternative because a single end-to-end profile would likely only capture the benchmark's average behavior, losing the latency characteristics of the individual pre-cluster–cluster pairs. This is undesirable because one of two scenarios may occur when the end-to-end data is applied to specific pre-cluster–cluster pairs. First, end-to-end data may fall short of potential speed up; second, end-to-end data may not identify the entire warm up region, sacrificing simulation accuracy.

In the first case, suppose the benchmark is partitioned into several pre-cluster–cluster regions that tend to access a small memory footprint in a tight loop. The resulting pre-cluster–cluster pairs will tend to require very small w_N to achieve an MRRL percentile of N. Suppose further that end-to-end, the benchmark occupies a much larger memory footprint whose access pattern appears sparse and more distributed; the w_N isolated from the end-to-end data would be much larger. This approach would warm up too many references, but even though speed up is suboptimal, IPC accuracy would remain intact.

In the second scenario, imagine that several pre-cluster–cluster partitions sparsely access a very large memory footprint requiring large w_N. The end-to-end memory reference behavior, however, tends to be less sparse resulting on average, in more frequent revisits to memory addresses than within the aforementioned pre-cluster–cluster pairs. If the end-to-end data are used to derive w_N for these pre-cluster–cluster pairs, the w_N will fall short of the true number of instructions necessary to achieve a MRRL percentile of N. While the simulations would run faster, IPC accuracy may falter. Hence, I designed *sim-mrrlprofile* to accept the same pre-cluster–cluster parameters as are given to *sim-outorder_mrrl*; each region is measured and dealt with individually.

While profiling with associative arrays did lead to the discovery of warm up intervals that yielded highly accurate IPC measurements, associative lookup was painfully slow. Sometimes profiling required more time than the simulations themselves! The problem is that reuse latency measurements are taken for every unique memory reference address encountered during a pre-cluster–cluster pair. Uniquely identifying each address requires a keyed associative lookup whose worst-case running time is linear in the number of unique addresses. This rapidly becomes overwhelming for pre-cluster–cluster pairs with large working sets. To rectify this, I mimicked true associativity with "deeply" associative arrays.

The principle is identical to set associativity in caches: while conflict-free unique reference identification is ideal, conflict-rarity is good enough in practice. With associativity, conflicts are reduced by borrowing set theory's notion of equivalence classes [24, 61] in the form of limited amounts of true associativity within each cache set among reference addresses with identical *cache index bits* [21].

In a finite machine such as defined by the Alpha AXP reference [52] with 32-bit addressing, the cardinality of each equivalence class is determined by the width of the cache index bit field and the cache block width. Thus, for n bits of index, if each cache block contains 2^k independently addressable units, there are 2^{32-n-k} elements in each equivalence class. In my deeply associative cache implementation, the low 2 bits of each reference address are discarded, yielding word-width uniqueness granularity. Then, the next lowest n bits are used as an index to hash to a location in the array. Finally, the highest $32 - (n + 2)$ bits are used as a tag to match against any of the a associative buckets. Since the associativity of these arrays is fixed (a), the worst-case lookup time is constant.

The first *sim-mrrlprofile* implementation used an index width (n) of 16 bits and 128 degrees of associativity (a). With 128 degrees of associativity (thus the title *deeply* associative), collisions between distinct reference addresses was a rare occurrence; unique addresses that hash to the same array element may occupy any one of 128 fully associative buckets. While this high associativity did very closely mimic a fully associative array, there was only a small speed-up over full associativity. I traced this problem to poor performance in the native cache on the host platform executing the profiler. In theory, worst-case linear search over a fixed domain of associativity (*i.e.*, 128 possibilities) is less expensive than worst-case linear search over virtually unlimited associativity (*i.e.*, 2^{30} possibilities). In practice however, when even fixed associativity leads to poor cache performance, code runs slowly. To rectify this, I drastically reduced the associativity to 32 degrees and quadrupled the number of array elements. This massive is new organization kept collisions at bay and did very well at mimicking full associativity.

The 75% associativity reduction brought less garbage into the native cache on failed lookups and yielded a handsome performance improvement. This final implementation of *sim-mrrlprofile* is the version used to acquire the dissertation results, and the version offered from the MRRL Web site. It enables very good warm phase identification (especially at the 99-th percentile) that yields IPCs that are faithful to *fullwarmup*'s.

6.2 Cycle-accurate Simulator: sim-outorder_mrrl

The primary goal of this research was to maintain the accuracy of IPC measurements for sampled simulation achieved from *fullwarmup*, but in less time. Hence, it was critically important to measure simulation running time as accurately as possible. To achieve high accuracy, the first modification to *sim-outorder* to create *sim-outorder_mrrl* was to use the UNIX system call *getrusage*() [41] to monitor the CPU time of each execution. CPU utilization measurement is engaged immediately after the benchmark binary is loaded into the simulated virtual address space and ends as soon as the main simulator loop exits. This modification enables accurate timing analysis of the simulations even when the host system is heavily loaded.

Further modifications were necessary to incorporate the three-phase (cold–warm–hot) simulation strategy described in Chapter 1; unlike the original *sim-outorder*, my modifications also accommodate multiple clusters by allowing the simulator to jump from the hot phase back to either the cold phase or the warm phase.

A third enhancement was necessary because the warm phase varies in size for different MRRL percentiles (N), as well as for *nowarmup* and *fullwarmup* causing the amount of cache and branch predictor modeling to vary accordingly. Effective analysis of cache and branch predictor statistics however, demands comparable measurements; since the size and position of each cycle-accurate sample cluster is fixed for a given benchmark these clusters are ideal for gathering cache and branch predictor statistics. To enable fined-grained manipulation of cache and branch predictor statistics gathering, code was added that facilitated engaging and disengaging statistics gathering at arbitrary points during a benchmark's simulation, for an arbitrary collection of cache and branch predictor statistics (*e.g.*, *cache_dl1*$\rightarrow$ *misses*, *pred*$\rightarrow$ addr_hits). This code was then used to engage cache and branch predictor statistics gathering exclusively during the hot phase; thus, cache and branch prediction data were gathered precisely the same as IPC: during the clusters. This ensured that these measurements were comparable among different simulator configurations, regardless of the amount of cache and branch predictor warm up.

Finally, there are subtle caveats that arose when developing the mechanism for fine-grained control over the branch predictor, instruction cache and data cache warm up. First, since the profiler tracks MRRL data for three different reference streams three different w_N are necessary: $w_{N_{branch}}$, $w_{N_{instruction}}$, $w_{N_{data}}$. When the simulator executes, branch predictor warm up is obviously engaged at $w_{N_{branch}}$ instructions prior to the hot phase. Elements of the cache hierarchy however, may not be entirely independent and therefore, require special treatment. Consider for example, a cache hierarchy composed of separate secondary caches for instructions and for data (just as is typically done for first-level caches). In this setup, data caching and instruction caching would be engaged independently, according to their respective w_Ns. (The same would be true of a cache hierarchy with independent first-level instruction- and data caches that foregos second-level caches.) As shown in Table 5.8, on the other hand, my experiments use a *unified* secondary cache, that houses both instructions and data. In this environment, both instruction caching and data caching must be engaged simultaneously, since engaging warm up of one reference stream before the other gives the first an unfair opportunity to become established in the L2. The typically tumultuous relationship between the two warring factions—instructions and data—would not be accurately modeled and may adversely influence their respective miss rates during the hot phase. Therefore, in simulations that model a unified L2 (and/or unified L3), instruction cache and data cache modeling are engaged simultaneously at whichever stream's w_N is farther from the start of the hot phase.

6.3 Related Work

SimpleScalar is a very popular and widely-used [36] tool for computer architecture research. In addition to *sim-inorder*, *sim-mrrlprofile* and *sim-outorder_mrrl*, numerous other cycle-accurate simulation tools have been built within the SimpleScalar framework including *sim-dmt*, to perform Differential Multithreading (dMT) research [15, 16]; Wattch [3], to perform microprocessor power analysis; and HydraScalar [53], to model multipath execution. SimpleScalar, however, is only one of many simulation tools. Others include the MIPS instruction-level simulator SPIM [34], the SimOS [23] toolkit, and the SMTSIM [69] simultaneous multithreading simulator. Proprietary simulation software includes ARDI's instruction-level Syn68k [26] which emulates binaries compiled into the Motorola 680x0 [14] instruction set, and AMD's SimNow! [1] instruction-level simulator which simulates AMD's new x86-64 instruction set.

Instruction-level simulators are not appropriate for cycle-accurate performance evaluation because they do not model the synchronous movement of instructions through a pipeline. Rather, instruction-level simulators merely emulate some instruction set architecture. Software-driven simulators such as SPIM, SimpleScalar's *sim-safe* perform this emulation by manipulating architected state iteratively, in a fetch–decode–execute loop one instruction at a time. Simulators such as Syn68k, on the other hand, use an advanced form of software instruction-level simulation called *binary translation* [5, 6, 26]. The Syn68k software works by first converting the instructions of a 680x0 binary into equivalent "synthetic opcodes" that can either be executed in a virtual machine or translated and executed as equivalent instructions on the host architecture. While often faster than iterative simulators, because they do not model pipeline state, binary translators are also inappropriate for cycle-accurate performance evaluation. However, instruction-level simulation was very well suited to the task of making reuse latency measurements because these measurements only require information about the completed instruction count and reference addresses—not pipeline state. MRRL's evaluation as a warm up technique, on the other hand, required cycle-accurate instruction throughput analysis. This immediately disqualified all instruction-level simulators. This explains why, *sim-mrrlprofile* was developed by by extending the instruction-level *sim-safe* tool, while *sim-outorder_mrrl* was developed by extending the more sophisticated *sim-outorder* pipeline simulator.

Another processes that is closely related to simulation involves applying modifications to the actual binary image of the benchmarks to be studied. Then rather than emulating the binary opcodes and modifying some amount of virtual processor state, the modified binary is executed natively. As the native execution proceeds, the appended routines execute alongside the original code, gathering statistics in response to specific microprocessor events. Research that studies data cache miss rates for instance, would modify a benchmark binary to trap to a routine that records the source/destination address of each load/store instruction. This address record can either be manipulated on-line, inside the recording routine, passed to another routine, or stored as a trace for off-line processing.

Srivastava and Eustace's ATOM [56] is a binary instrumentation tool that facilitates customized program analysis. ATOM allows analysis code to be inserted anywhere inside a binary by modifying the object files of the target benchmark. Once these manipulations have been completed, the object files are linked into an *instrumented* binary. Research that requires knowledge about basic block execution frequency (such as the tools used in Sherwoord *et al.*'s research [50, 51]) would insert code prior to every basic block that traps to a routine to increment a counter corresponding to each basic block.

Larus and Schnarr's Executable Editing Library (EEL) [35] is similar to ATOM, but directly modifies fully linked binaries. EEL is able to both remove code and insert additional code that observes or alters a benchmark's execution. EEL provides a collection of high-level abstractions that hide specifics such of the hardware instruction set, binary file formats, and the reorganization of hard-coded references after binary modifications have been applied. Two of EEL's key abstractions are *control-flow graphs* (CFGs) and *snippets*. A CFG maps a binary's execution paths as a directed graph whose nodes correspond to basic blocks, and whose edges correspond to branches. A tool build with EEL modifies a binary by either deleting instructions, or appending code snippets to its CFG.

Binary modification is a useful tool for microarchitecture research, and in general has a tremendous speed advantage over software binary emulation, because of native hardware execution. Unfortunately, precisely that which is responsible for its speed advantage causes serious drawbacks. First, non-native binaries cannot be studied. (Researchers with x86-only computing resources cannot study 680x0 binaries.) Second, only those instructions that actually *commit* in the pipeline of the native host will be seen. This is because most modern microprocessors employ *speculation*. By predicting the destination of branch instructions that have not yet been resolved, the native processor is able to keep many more partially executed instructions in the pipeline. When these partially executed instructions finally complete, their results are buffered until the outcome of the preceding branch is know. If the processor guested its destination correctly, then the buffered results are committed to the actual hardware state; otherwise, the buffered results are discarded and execution restarts along the opposite path of the offending branch. Since only perfect pipeline state is captured, detailed pipeline analysis is impractical with binary instrumentation.

Chapter 7

Contributions, Conclusion & Future Work

Software simulation is a flexible tool for microarchitecture research. The tremendous slowdown factor of cycle-accurate simulation relative to native hardware execution, however, has driven the microarchitecture research community to search for methods to accelerate this process. One popular acceleration approach interleaves slow cycle-accurate simulation with much faster, low-detail simulation. If one or more sample clusters from the dynamic instruction stream can be identified that reliably mimic the end-to-end execution of the entire benchmark, then only this sample of clusters needs to be simulated in cycle-accurate detail.

Cache and branch prediction profoundly impact the cycle-by-cycle movement of instructions through the pipeline (*i.e.*, instruction throughput) [21]. Hence, reliable performance measurements from sampled simulation requires that the simulated cache hierarchy and branch predictor perform during the sample clusters precisely or at least approximately as they would have if the entire benchmark were simulated in cycle-accurate detail. This is the responsibility of warm up. One way to guarantee accurate cache and branch predictor state is to simulate non-sample instructions using *fullwarmup*, which augments functional simulation by modeling the cache and branch predictor interactions required for instruction fetch, load/store instructions, and control flow instructions. While substantially faster than cycle-accurate simulation, *fullwarmup* is still slow and can be prohibitive if research requires large state-space searches.

This dissertation demonstrates that *fullwarmup* is often unnecessary and offers techniques for selecting a smaller warm up phase that still preserves simulation accuracy. The major contributions of this research are

1. **MSE**—a rigorous mathematical framework for quantitatively reasoning about the probability of touching a certain proportion of cache blocks based solely on the dimensions of the cache and the number of unique memory references handled in the cache. *MSEwarmup* adapts this framework, forging a technique for accurate, rapid warm up of L1 cache state.

2. **MRRL**—a more direct approach to warm up that builds upon MSE by exploiting temporal locality. Rather than warm up a specific proportion of cache blocks, MRRL seeks to warm up specifically those blocks that will be accessed during the sample clusters. MRRL is useful for warming up any cache hierarchy

organization and well as branch predictors.

7.1 Minimal Subset Evaluation

Although *MSEwarmup* is superseded by MRRL, Minimal Subset Evaluation (MSE) is no less a valuable contribution in its own right. MSE's contribution stems from its original use of combinatorics, probability, and statistical methods to *quantitatively articulate the probability of touching a certain proportion of cache blocks* based solely upon the dimensions of the cache and the count of unique memory reference addresses handled within the cache. As discussed in Chapter 1, before MSE, several techniques were developed [7, 8, 28, 42] for defeating cold-start bias more efficiently than *fullwarmup*. Except for Nguyen's PARSIM [42], these were empirical techniques. Unfortunately however, in addition to other cumbersome requirements, PARSIM requires *a priori* knowledge of the overall cache miss rate and the load/store density of the dynamic instruction stream, which implies at least a one-time cost *fullwarmup* run to measure it.

The MSE research yielded the MSE formula for computing the probability that mapping m unique references into the cache will touch $\alpha N \beta a$ cache blocks and a tractable approximation to make this calculation quickly. These are the basis of the quantitative framework and the first contribution of MSE. Furthermore, as discussed in Chapter 3, the MSE formula depends upon the distribution of unique references throughout the cache; this required that I determine and verify this distribution and is MSE's second contribution: the application of statistical methods to study the distribution of unique memory reference address mappings inside the cache.

MSEwarmup adapts MSE by using the MSE formula to compute the number of unique references, m, necessary to touch a specific proportion of L1 cache blocks with user-chosen probability, p. By profiling the pre-cluster instructions, *MSEwarmup* determines the number of instructions prior to the sample that contains m unique reference addresses. If m or more uniques can be found among the pre-cluster instructions, *MSEwarmup* begins warm up late in the pre-cluster phase. If fewer than m uniques are among the pre-cluster instructions, *MSEwarmup* degenerates to *fullwarmup*, trading away speed up for accuracy.

In practice (see Chapter 5), MSE yields highly accurate simulations while substantially reducing simulation times. For $p = 99.9\%$, where p is the user-chosen probability of touching a certain proportion of cache blocks, the average error in IPC relative to the *fullwarmup* IPC was 0.3% and the average reduction in simulation running time was 47%; for $p = 95.0\%$, the average error in IPC was 0.4% and the average reduction in simulation running time was 61%.

7.2 Memory Reference Reuse Latency

While MRRL builds upon MSE, it does so by approaching the warm up problem very differently, seeking to touch specifically those blocks that matter to the sample clusters. MRRL's key insight is that *temporal locality is an effective guide for pre-cluster warm up.* MRRL quantifies temporal locality as a percentile of consecutive access latencies among all references in a pre-cluster–cluster pair. For percentile N

= 100%, MRRL immediately bounds the size of the warm phase. This is because at $N = 100\%$, *all* references require w_N or fewer instructions between consecutive accesses; $w_{1.000}$, therefore is the *maximum reuse latency.* Hence, it is pointless to model references that occur more than $w_{1.000}$ instructions prior to a sample cluster. Results from Chapter 5 however, show that in practice, $N = 99.0\%$ works well, achieving good accuracy and substantial speed up.

The enumerated list below discusses how MRRL meets the research objectives.

1. **Any Cache Hierarchy.** MRRL is able to determine the beginning of a warm phase which will simultaneously ensure accurate warm up for all levels of the cache hierarchy regardless of their organization. This is critically important as deeper levels of cache may become commonplace in future systems. (The IBM POWER4 [65] for instance, supports a third level of cache.) Since MRRL is unaffected by cache hierarchy depth, as well as the organization of a cache's constituent levels (*e.g.*, block width, associativity, unified or separate hosting of instructions and data), this allows—per benchmark–input pair and sampling regime—only a one-time profiling cost that can be used to obtain warm up points for any simulated cache design. This makes MRRL a more available and flexible warm up strategy than *MSEwarmup* for contemporary and forward-looking designs.

2. **Branch Predictors.** *MSEwarmup* is not well-suited for branch predictor warm up, because large branch predictor buffers [21] require a very large number of

unique branch addresses to warm up with high (*e.g.*, $p = 99.9\%$) accuracy. In the absence of a technique to accurately gauge the MSE tuning variable α, MSE for such a large number of uniques quickly degenerates to *fullwarmup*, maintaining accuracy but completely trading away speed. On the other hand, MRRL's alternative warm up approach exploits reuse latency characteristics and the analogy between program counter variation and recurrent fractal random walks [66], and presents a suitable vehicle for warming up the 2-bit saturating counters of branch predictor buffers.

3. **Sampling Regime.** Experimental results shown in Chapter 5 that deploy MRRL in uniform systematic sampling, random cluster sampling, and samples drawn by more sophisticated methods ([50, 51]), demonstrate MRRL's independence from sampling strategy. (This is a trait shared with *MSEwarmup*.) Random cluster sampling's specific amenability to rigorous statistical analysis was exploited to demonstrate that at the 99.0-th percentile, MRRL achieves instruction throughput measurements that are statistically identical to *fullwarmup*, while simulating in much less time than *fullwarmup*.

Results in Chapter 5 show that for the 99.0-th percentile ($\text{MRRL}_{0.990}$), MRRL generated an average error in IPC relative to *fullwarmup*, of less than 0.5% for the simple equidistantly-spaced simulation sample methodology, and less than 0.4% for randomly-chosen cluster samples, while the 99.9-th percentile generated an average IPC error of less than 0.3% for automatically-chosen (BBDA) samples [51], (with 2-

and 3-level cache). In the equidistant-cluster experiments $\mathrm{MRRL}_{0.990}$ simulated in roughly 70% the time of *fullwarmup*, translating into an average of roughly 97% of the maximum potential speed up. Further experiments with Sherwood *et al.*'s [50, 51] basic-block distribution analysis (BBDA) samples show a percent-error in IPC measurements of less than 0.03% in all cases for $\mathrm{MRRL}_{1.000}$ relative to *fullwarmup*. This $\mathrm{MRRL}_{1.000}$ however, while very reliable, does not speed up simulation as effectively as for $N \in (0,1)$. Five benchmarks saw no improvement, but on average $\mathrm{MRRL}_{1.000}$ achieved roughly 33% of the maximum possible speed up. Finally, to prove its flexibility and one-time profiling cost, I applied MRRL to the same BBDA samples, but simulated a radically different cache organization with three levels—two of which were unified—and different block widths on each level. Since this last round of experiments used the same samples, warm phases were chosen from the original BBDA profiles. Experimental data prove that in spite of a very different cache organization, the same MRRL profiles produce highly accurate throughput data, achieving a maximum percent error of 1.05% for $\mathrm{MRRL}_{0.999}$, and an average simulation time reduction of 45% relative to *fullwarmup*.

7.3 Future Work

As discussed in Chapter 5, in the statistical analysis of *MSEwarmup* and MRRL, the random cluster sampling experiments used a cluster size of 1 million instructions. This was a conservative adjustment to the 100,000-instruction clusters used by Conte *et al.* [8], that allowed more pipeline behavior per cycle-accurate sample cluster to be observed and measured. One avenue for future research would be to explore the space of cluster sizes when applied to random cluster sampling, and to observe the impact of cluster size on sample accuracy for a given sample size. One experiment for instance, could use a sample size of 50 million instructions, measured as 500 $\times$ 100,000 instructions, versus 50 $\times$ 1 million instructions, versus 5 $\times$ 10 million instructions. Additional experiments that vary the sample size could also be organized. A statistical test such as ANOVA—analysis of variance—could then be used to characterize the amount of difference between the various sampling approaches.

Another avenue for future research would be collaboration with Timothy Sherwood and colleagues at the University of California at San Diego, to combine the Basic Block Distribution Analysis (BBDA) technique [50, 51] with MRRL; BBDA would be used to find suitable sample clusters, and MRRL to find short warm up intervals. Sherwood's preliminary results show only small errors for a typical 2-level cache organization when *stale-state/nowarmup* is used. As demonstrated in Chapter 5 however, this inadequate warm up strategy yielded greater error when a third level of cache was added (which increased the main memory access latency). While this dissertation contains $\text{MRRL}_{0.999}$ warm up points for the 100-million-instruction BBDA sample clusters (see Appendix A), it remains to explore the space of cluster sizes with BBDA, and differing microarchitecture configurations. Also interesting, would be BBDA-guided stratified sampling: random cluster sampling within the clusters chosen by BBDA. Combining MRRL with BBDA would not only accelerate the joint research, but if applied to the SPEC CPU2000 benchmarks, the research results would have a positive impact on the large body of ongoing research that uses this benchmark suite.

Acknowledgments

First, a special word of thanks my wife Latitia M. Haskins for good council, patience, and answers to countless computer engineering questions; and to my daughter Jasmine B. Haskins for teaching patience to me. Thank you Jewel, Junell, and John, Sr. for 20+ years of encouragement. Thank you Prof. Kevin Skadron for interviewing at UVA, accepting UVA's employment offer, and ultimately accepting me as your first Ph.D. student. I would also like to thank Dr. Dee A. B. Weikle, Prof. Margaret Martonosi, Prof. Tom Conte, Dr. Jesse Bemley, Dr. Duncan Buell, and the anonymous reviewers of submitted papers for their valuable feedback and insights. Thanks also to the founders of the Ballard Fellowship. Many thanks also to Virginia Hilton, Brenda Perkins, and Tammy Ramsey for protecting me from myself all throughout graduate school. A computer scientist is next to nothing without smoothly operating computers, and therefore a tremendous debt of thanks goes to the Department's talented systems staff. And finally, for more than 27 years I have been divinely blessed, led, and inspired; hence, I give preeminent thanks to the Almighty.

This material is based upon work supported in part by the National Science Foundation under grant no. CCR-0082671.

Bibliography

[1] Advanced Micro Devices. AMD SimNow! simulator. Web site, http://www.x86-64.org/downloads/.

[2] T. M. Austin. SimpleScalar home page. Computer Software, http://www.simplescalar.com/.

[3] D. Brooks and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the International Symposium on Computer Architecture*, Jun 2000.

[4] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *Computer Architecture News*, 25(3):13–25, June 1997.

[5] C. Cifuentes and M. Van Emmerik. UQBT: adaptable binary translation at low cost. *IEEE Computer*, 33(3):60–66, 2000.

[6] C. Cifuentes, M. Van Emmerik, N. Ramsey, and B. Lewis. Experience in the design, implementation and use of a retargetable static binary translation framework. Technical Report TR-2002-105, Sun Microsystems Laboratories, Jan. 2002.

[7] M. Co and K. Skadron. The Effects of Context-Switching on Branch Prediction Performance. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, Nov. 2001.

[8] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing State Loss for Effective Trace Sampling of Superscalar Processors. In *Proceedings of the International Conference on Computer Deisgn*, Oct. 1996.

[9] Standard Performance Evaluation Corporation. SPEC CPU2000 press release faq, 2000. WWW site: http://www.specbench.org/osg/cpu2000/press/faq.html.

[10] H. J. Curnow and B. A. Wichmann. A synthetic benchmark. *The Computer Journal*, 19(1):43–49, Feb. 1976.

[11] M. C. Easton and R. Fagin. Cold-start versus warm-start miss ratios. *Communications of the ACM*, 21(10):866–871, Oct. 1978.

[12] E. N. Elnozahy. Address trace compression through loop detection and reduction. In *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 214–15, May 1999.

[13] J. E. Freund. *Mathematical Statistics*. Prentice-Hall, Inc., 1971.

[14] T. L. Harman and D. T. Hein. *The Motorola MC68000 Microprocessor Family: Assembly Language, Interface Design and System Design*. Prentice Hall, 2nd edition, 1995.

[15] J. W. Haskins, Jr., K. Hirst, and K. Skadron. Inexpensive throughput enhancement in small-scall embedded microprocessors with block multithreading: Extensions, characterization, and tradeoffs. In *Proceedings of the International Performance Computing, and Communications Conference*, Apr. 2001.

[16] J. W. Haskins, Jr., and K. Skadron. Differential Multithreading: Recapturing Pipeline Stall Cycles and Enhancing Throughput in Small-Scale Embedded Microprocessors. In *Proceedings of the Workshop on Complexity-Effective Design*, Jun. 2000.

[17] J. W. Haskins, Jr., and K. Skadron. Minimal Subset Evaluation: Rapid Warm-up for Simulated Hardware State. In *Proceedings of the International Conference on Computer Design*, Sept. 2001.

[18] J. W. Haskins, Jr., and K. Skadron. Memory Reference Reuse Latency: Accelerated Sampled Microprocessor Simulation. Technical Report CS-2002-19, Univ. of Virginia Dept. of Computer Science, Jul. 2002.

[19] J. W. Haskins, Jr., and K. Skadron. Memory reference reuse latency: Accelerated warmup for sampled microarchitecture simulation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, Mar. 2003.

[20] J. W. Haskins, Jr., K. Skadron, AJ KleinOsowski, and D. L. Lilja. Techniques for Accurate, Accelerated Processor Simulation: Analysis of Reduced Inputs and

Sampling. Technical Report CS-2002-01, Univ. of Virginia Dept. of Computer Science, Jan. 2002.

[21] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers, Inc., San Mateo, California, 1995. Second Edition.

[22] G. T. Henry. *Practical Sampling*. SAGE Publications, Inc., 1990.

[23] S. Herrod, E. Bugnion, and S. Devine. SimOS: the complete machine simulator. Computer Software, http://simos.stanford.edu/.

[24] I. N. Herstein. *Abstract Algebra, Second Edition*. MacMillan Publishing Company, 1990.

[25] T. Horel and G. Lauterbach. UltraSPARC-III: Designing third-generation 64-bit performance. *IEEE Micro*, pages 73–84, May-June 1999.

[26] M. Hostetter. Syn68k: ARDI's dynamically compiling 68LC040 emulator. Downloadable White Paper, ftp://ftp.ardi.com/pub/SynPaper.ps, Oct 1995.

[27] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th International Symposium on Computer Architecture*, June 2001.

[28] R. E. Kessler, M. D. Hill, and D. A. Wood. A Comparison of Trace-Sampling Techniques for Multi-Megabyte Caches. Technical Report 1048, Univ. of Wisconsin-Madison Computer Sciences Dept., September 1991.

[29] AJ KleinOsowski, J. Flynn, N. Meares, and D. J. Lilja. Adapting the SPEC 2000 Benchmark Suite for Simulation-Based Computer Architecture Research. In *Proceedings of the Third IEEE Annual Workshop on Workload Characterization*, pages 73–82, Sep. 2000.

[30] AJ KleinOsowski and D. J. Lilja. MinneSPEC: A new spec benchmark workload for simulation-based computer architecture research. *IEEE Computer Architecture Letters*, 1, Jun 2002.

[31] T. Lafage and A. Seznec. Choosing Representative Slices of Program Execution for Microarchitecture Simulations: A Preliminary Application to the Data Stream. In *Proceedings of the Third IEEE Annual Workshop on Workload Characterization*, pages 102–110, Sep. 2000.

[32] S. Laha, J. H. Patel, and R. K. Iyer. Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems. *IEEE Trans. Computers*, 37(11):1325–1336, November 1988.

[33] A. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proc. ISCA-28*, Jun. 2001.

[34] J. Larus. SPIM home page. Computer Software, http://www.cs.wisc.edu/ larus/spim.html.

[35] J. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, June 1995.

[36] SimpleScalar LLC. Frinds of SimpleScalar LLC. Web site, http://www.simplescalar.com/friends.html.

[37] B. B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman & Company, 1983.

[38] F. H. McMahon. The livermore fortran kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Dec 1986.

[39] A. Mendelson, D Thiébaut, and D. K. Pradhan. Modeling live and dead lines in cache memory systems. *IEEE Trans. Computers*, 42(1):1–14, Jan. 1993.

[40] MIPS Technologies. *MIPS R10000 Microprocessor User's Manual*, June 1995. Version 2.0.

[41] M. Mitchell, A. Samual, and J. Oldham. *Advanced Linux Programming*. Pearson Education, 2001.

[42] A. Nguyen, J. Wellman, and P. Bose. PARSIM: a parallel trace-driven simulation facility for fast and accurate performance analysis studies. In *Proceedings of the International Performance Computing, and Communications Conference*, Apr. 1997.

[43] K. M. Obenland. *Using Simulation to Assess the Feasibility of Quantum Computing.* PhD thesis, University of Southern California, Aug. 1998.

[44] K. M. Obenland and A. M. Despain. A parallel quantum computer simulator. In *Proceedings of High Performance Computing '98*, Apr. 1998.

[45] D. L. Perry. *VHDL: Programming by Example.* McGraw-Hill Professional, 4th edition, 2002.

[46] C. Price. *MIPS IV Instruction Set, Revision 3.1.* MIPS Technologies, Inc., Mountain View, CA, Jan. 1995.

[47] T. Shanley. *Pentium Pro Processor System Architecture.* Addison Wesley Professional, Boston, MA, 1996.

[48] H. Sharangpani and K. Arora. Itanium processor microarchitecture. *IEEE Micro*, pages 24–43, Sept.–Oct. 2002.

[49] T. Sherwood and B. Calder. Spec 2000 simpoint error analysis using basic block vectors. Online Article: http://www-cse.ucsd.edu/ calder/simpoint/points/original/simpoint_error.html.

[50] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, Sept. 2001.

[51] T. Sherwood, E. Perelman, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[52] R. L. Sites and R. T. Witek. *Alpha AXP Architecture Reference Manual*. Digital Press, 2nd edition, 1995.

[53] K. Skadron and P. S. Ahuja. HydraScalar: a multipath-capable simulator. In *Newsletter of the IEEE Technical Committee on Computer Architecture*, Jan 2001.

[54] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Branch prediction, instruction-window size, and cache size: Performance tradeoffs and simulation techniques. *IEEE Trans. Computers*, 48(11):1260–81, Nov. 1999.

[55] P. Song. UltraSparc-3 aims at MP servers. *Microprocessor Report*, pages 29–34, Oct. 27 1997.

[56] A. Srivastava and A. Eustace. ATOM A System for Building Customized Program Analysis Tools. In *Proc. PLDI 1994*, June 1994.

[57] Standard Performance Evaluation Corporation. SPEC CPU2000 Benchmarks. WWW site: http://www.specbench.org/osg/cpu2000, Dec. 1999.

[58] Standard Performance Evaluation Corporation. SPEC CPU95 Benchmarks. WWW site: http://www.specbench.org/osg/cpu95, Dec. 1999.

[59] M. Sternstein. *Statistics*. Barron's Educational Series, Inc., 1996.

[60] J. Stokes. Behind the benchmarks: SPEC, GFLOPS, MIPS *et al.* Online Article: http://www.arstechnica.com/cpu/2q99/benchmarking-1.html.

[61] R. R. Stoll. *Set Theory and Logic*. Dover Publications, Inc., 1979.

[62] W. D. Strecker. Transient behavior of cache memories. *ACM Trans. on Computer Systems*, 1(4):281–293, Nov. 1983.

[63] K. Subramani, M. Chetlur, T. J. McBrayer, R. Radhakrishnan, and P. A. Wilsey. TyVIS: A VHDL simulation kernel (documentation for version 1.0). Online Documentation: http://www.ececs.uc.edu/ paw/tyvis/doc/tyvis.html.

[64] J. R. Taylor. *An Introduction to Error Analysis: The Study of Uncertainty in Physical Measurements*. University Science Books, 1982.

[65] J. M. Tendler, J. S. Dodson, Jr. J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, pages 5–25, Jan. 2002.

[66] D. Thiébaut. On the fractal dimension of computer programs and its application to the prediction of the cache miss ratio. *IEEE Trans. Computers*, 38(7):1012–1026, Jul. 1989.

[67] D. Thiébaut and H. S. Stone. Footprints in the cache. *ACM Trans. on Computer Systems*, 5(4):305–329, Nov. 1987.

[68] D. Thiébaut, J. L. Wolf, and H. S. Stone. Synthetic traces for trace-driven simulation of cache memories. *IEEE Trans. Computers*, 41(4):388–410, Apr. 1992.

[69] D. M. Tullsen. Simulation and modeling of a simultaneous multithreaded processor. In *22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[70] B. Underwood, C. Duncan, J. Taylor, and J. Cotton. *Elementary Statistics.* Appleton-Century-Crofts, 1954.

[71] R. P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, Oct. 1994.

[72] D. A. Wood, M. D. Hill, and R. E. Kessler. A Model for Estimating Trace-Sample Miss Ratios. In *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 79–89, June 1991.

[73] S. Yalamanchili. *VHDL Starter's Guide.* Prentice Hall, 1997.

Appendix A

MRRL$_{0.999}$ Warm Up Points for BBDA

This appendix give the data gathered as a collaborative research between Dr. Skadron and me, and Timothy Sherwood and his colleagues at the University of California at San Diego. Our research combines the strength of Basic Block Distribution Analysis [51] to find relevant samples (called *simulation points*) and the strength of Memory Reference Reuse Latency to accelerate the establishment of accurate pre-cluster state. Together, this research seeks to offer guidance to the computer architecture research community for performing cycle-accurate simulation with the SPEC CPU2000 benchmark suite quickly and accurately. The material is organized as a table and gives the warm up points for MRRL at the 99.9-th percentile. The first column gives the warm up point for instructions; the second column gives the warm up point for data; the third column gives the warm up point for branches; and the final column gives the simulation points gathered from BBDA. In other words, for each BBDA simulation point, the warm up points gives the number of cold phase instructions for instruction, data, and branch prediction. Once the cold phase completes, warm up begins, and when the total number of completed instructions (cold and warm) reaches the number prescribed by the simulation point, cycle-accurate simulation begins.

Chapter 5 shows that MRRL with $N = 0.999$ is very successful at achieving accurate, rapid warm up. Recall that reuse latency profiles are gathered by partitioning the discrete interval $[1, l]$ (which is bijectively mapped to the pre-cluster–cluster instructions; see Chapter 4) into $n \ll l$ bucket$_i$s, where each bucket$_i$ represents some subset $[a, b]$ of $[1, l]$. As the profile proceeds, it counts the number of completed instructions between consecutive accesses to each unique reference addresses, say C, and increments the count of the bucket$_i$ representing the subset that contains C. If $N\%$ of MRRL measurements require w_N instructions between consecutive accesses, then the MRRL-prescribed warm phase begins w_N instructions before the sample; this is MRRL warm up at the N-th percentile.

For many benchmarks, the 99.9-th percentile of all reuse latencies for instructions, data, and branches occurred in bucket$_1$. bucket$_1$ represents the interval subset $[1, b]$ and therefore, counts the occurrence of references that require from 1 to b instructions between consecutive accesses, *i.e.*, those references with the shortest reuse latencies. When all three (instructions, data, and branches) are accommodated in bucket$_1$, all three warm up points are identical for each simulation sample. This occurs for many of the warm up points for many of the benchmarks. Benchmarks that diverge from this uniformity for one or more samples include *gcc_200*, *gcc_expr*, and *bzip2_graphic*.

benchmark	instruction warm up	data warm up	branch warm up	simulation point
ammp	59441894404	59441894404	59441894404	59500000000
	106753808530	106753808530	106753808530	106800000000
	160647363154	160647363154	160647363154	160700000000
	177383691279	177383691279	177383691279	177400000000
	212765429624	212765429624	212765429624	212800000000
	243669824155	243669824155	243669824155	243700000000
	247995800654	247995800654	247995800654	248000000000
	302546679624	302546679624	302546679624	302600000000
	311191601499	311191601499	311191601499	311200000000
applu	62339062500	62339062500	62339062500	62400000000
	137926171875	137926171875	137926171875	138000000000
	150687597529	150687597529	150687597529	150700000000
	162488476499	162488476499	162488476499	162500000000
	195567675654	195567675654	195567675654	195600000000
	223372851499	223372851499	223372851499	223400000000
apsi	89512500000	89512500000	89512500000	89600000000
	100689160029	100689160029	100689160029	100700000000
	161740331904	161740331904	161740331904	161800000000
	210652246030	210652246030	210652246030	210700000000
	286226171875	286226171875	286226171875	286300000000
bzip2_graphic	10589648374	10589648374	10589648374	10600000000
	14795898374	14795898374	14795898374	14800000000
	16798046875	16794140625	16798046875	16800000000
	19497363154	19494726436	19497363154	19500000000
	42977050654	42977050654	42977050654	43000000000
	51891308530	51891308530	51891308530	51900000000
	76176269404	76176269404	76176269404	76200000000
	87189257749	87189257749	87189257749	87200000000
	104183398374	104183398374	104183398374	104200000000
	143461621030	143461621030	143461621030	143500000000

benchmark	instruction warm up	data warm up	branch warm up	simulation point
bzip2_program	7792382749	7792382749	7792382749	7800000000
	9398437500	9393750000	9398437500	9400000000
	13995507749	13995507749	13995507749	14000000000
	34080371030	34080371030	34080371030	34100000000
	44489843750	44489843750	44489843750	44500000000
	46797753779	46795507686	46797753779	46800000000
	60586523374	60586523374	60586523374	60600000000
	85875292905	85875292905	85875292905	85900000000
	98987206904	98987206904	98987206904	99000000000
	100498535029	100492675657	100498535029	100500000000
bzip2_source	6393750000	6393750000	6393750000	6400000000
	17688964780	17688964780	17688964780	17700000000
	39478710874	39478710874	39478710874	39500000000
	48790917905	48790917905	48790917905	48800000000
	51097753779	51097753779	51097753779	51100000000
	52998144404	52998144404	52998144404	53000000000
crafty	12287988154	12287988154	12287988154	12300000000
	50962206904	50962206904	50962206904	51000000000
	66384960874	66384960874	66384960874	66400000000
	112255175654	112255175654	112255175654	112300000000
equake	299706904	299706904	299706904	300000000
	6194238154	6194238154	6194238154	6200000000
	33573242124	33573242124	33573242124	33600000000
	46287597529	46287597529	46287597529	46300000000
	87359863154	87359863154	87359863154	87400000000
	129159179624	129159179624	129159179624	129200000000
facerec	34766015625	34766015625	34766015625	34800000000
	139697460874	139697460874	139697460874	139800000000
	152787304624	152774609312	152787304624	152800000000
	193460253779	193460253779	193460253779	193500000000
	197595996030	197547949158	197595996030	197600000000

benchmark	instruction warm up	data warm up	branch warm up	simulation point
fma3d	4695410029	4695410029	4695410029	4700000000
	11193652280	11193652280	11193652280	11200000000
	20890527280	20890527280	20890527280	20900000000
	50870703125	50870703125	50870703125	50900000000
	84167480405	84167480405	84167480405	84200000000
	159925976499	159925976499	159925976499	160000000000
galgel	51549609375	51549609375	51549609375	51600000000
	101651074155	101651074155	101651074155	101700000000
	207996093750	207996093750	207996093750	208100000000
	214094140625	214094140625	214094140625	214100000000
	216098046875	216098046875	216098046875	216100000000
	218098046875	218098046875	218098046875	218100000000
	346474511655	346474511655	346474511655	346600000000
	351095605405	351095605405	351095605405	351100000000
gcc_200	799218750	798437500	799218750	800000000
	57444628779	57444628779	57444628779	57500000000
	58698828125	58696484375	58698828125	58700000000
	92067382749	92067382749	92067382749	92100000000
	101091210874	101091210874	101091210874	101100000000
gcc_expr	899121030	898242124	899121030	900000000
	2498437500	2496875000	2498437500	2500000000
	4198339780	4198339780	4198339780	4200000000
	6297949155	6295898374	6297949155	6300000000
	8098242124	8098242124	8098242124	8100000000
	8799316279	8795214721	8799316279	8800000000
gzip_graphic	99902280	99902280	99902280	100000000
	8691601499	8691601499	8691601499	8700000000
	37272070249	37272070249	37272070249	37300000000
	46091406250	46091406250	46091406250	46100000000
	56589746030	56589746030	56589746030	56600000000
	96061425654	96061425654	96061425654	96100000000
gzip_program	22777734375	22777734375	22777734375	22800000000
	47176171875	47176171875	47176171875	47200000000
	59388085874	59388085874	59388085874	59400000000
	77881933530	77881933530	77881933530	77900000000
	140938378779	140938378779	140938378779	141000000000

benchmark	instruction warm up	data warm up	branch warm up	simulation point
gzip_source	16683691279	16683691279	16683691279	16700000000
	24792089780	24792089780	24792089780	24800000000
	32692285029	32692285029	32692285029	32700000000
	37295507749	37295507749	37295507749	37300000000
	65572363154	65572363154	65572363154	65600000000
	71993750000	71993750000	71993750000	72000000000
lucas	45755273374	45755273374	45755273374	45800000000
	52393554624	52393554624	52393554624	52400000000
	60192382749	60192382749	60192382749	60200000000
	98162890625	98162890625	98162890625	98200000000
	136962109375	136962109375	136962109375	137000000000
mesa	39761132749	39761132749	39761132749	39800000000
	97643456904	97643456904	97643456904	97700000000
	184515136655	184515136655	184515136655	184600000000
	280506250000	280506250000	280506250000	280600000000
mgrid	4295800654	4295800654	4295800654	4300000000
	80625390625	80625390625	80625390625	80700000000
	247437011655	247437011655	247437011655	247600000000
	310938085874	310938085874	310938085874	311000000000
	345865917905	345865917905	345865917905	345900000000
parser	176927050654	176927050654	176927050654	177100000000
	200776855405	200776855405	200776855405	200800000000
	334069726499	334069726499	334069726499	334200000000
	477060351499	477060351499	477060351499	477200000000
	510167773374	510167773374	510167773374	510200000000
swim	3796288999	3796288999	3796288999	3800000000
	70934375000	70934375000	70934375000	71000000000
	77693456904	77693456904	77693456904	77700000000
	194985351499	194985351499	194985351499	195100000000
	210085351499	210085351499	210085351499	210100000000
twolf	31169531250	31169531250	31169531250	31200000000
	96036621030	96036621030	96036621030	96100000000
	205293261655	205293261655	205293261655	205400000000
	288718554624	288718554624	288718554624	288800000000
	326762890625	326762890625	326762890625	326800000000

benchmark	instruction warm up	data warm up	branch warm up	simulation point
vortex_lendian2	35964843750	35964843750	35964843750	36000000000
	39696386655	39696386655	39696386655	39700000000
	55384667905	55384667905	55384667905	55400000000
	63492089780	63492089780	63492089780	63500000000
	75188574155	75188574155	75188574155	75200000000
	92982617124	92982617124	92982617124	93000000000
wupwise	9091113154	9091113154	9091113154	9100000000
	152260058530	152260058530	152260058530	152400000000
	181071972529	181071972529	181071972529	181100000000
	305378515625	305378515625	305378515625	305500000000

Appendix B

MSE Unique Reference Address Profiles.

This appendix contains plots of the unique reference address distributions for each of the SPECInt95 benchmark results given in Tables 5.2 through 5.5. Recall that MSE is an analytical framework that allows one to quantitatively assess the likelihood that m unique reference addresses will touch some proportion of blocks within a cache. *MSEwarmup* adapts MSE by using its quantitative assessment capability to choose a point prior to each sample cluster to begin warm up. Starting from this point, the warm phase will encounter at least m unique reference addresses and will therefore have (user-chosen) probability p of touching $\alpha N \beta a$ cache blocks, where N is the number of cache sets, a is the degrees of associativity, and α, $\beta \in (0, 1]$ are tuning variables that indicate the proportion of sets and blocks per set, respectively.

Since *MSEwarmup* depends on witnessing a certain MSE-prescribed number (m) of unique references, its ability to speed up simulation depends heavily on the distribution of unique references among the pre-cluster instructions. Figures B.1 through B.6 each plot the distribution of unique references for a single benchmark. Each point on the x-axis corresponds to some MSE-prescribed number of hundreds of unique reference addresses (*i.e.*, for some x, $m = 100x$). The logarithmic y-axis gives the number of cold phase instructions. The length of the entire pre-cluster phase is given by the value of y at $x = 0$; thus, the length of the warm phase is computed as the difference of the entire pre-cluster length and the cold phase length: $y_{x=0} - y_{x=\frac{m}{100}}$.

For a pre-cluster–cluster pair of fixed length, the smaller the cold phase, the longer the warm phase (which leads to a long simulation time). This elongation of the warm phase due to a brief cold phase is illustrated vividly by the plot for the *go* benchmark whose pre-cluster phase lasts for nearly $y = 10^9$ instructions (see Figure B.3 at $x = 0$). As Figure B.3 shows, the length of the cold phase drops sharply, nearly three orders of magnitude after $x > 350$, to $y = 10^6$—a mere $\frac{1}{1000}$-th of *go*'s pre-cluster length. This is the phenomenon described in Chapter 4 as a *front-loaded* unique reference distribution. Since a very large number of warm phase instructions are necessary to witness the MSE-prescribed m uniques, *MSEwarmup*'s ability to speed up the simulation is substantially reduced.

Front-loading of unique references does not occur for the other five benchmarks (Figures B.1, B.2, and B.4 through B.6). Accordingly, their unique reference distribution plots stay level, dropping only slightly for increasing x. This means that the cold phase can be nearly as long as the entire pre-cluster phase, while the short warm phase still witnesses the MSE-prescribed m unique references. Because of the latter, *MSEwarmup* is able to speed up these benchmarks' simulations.

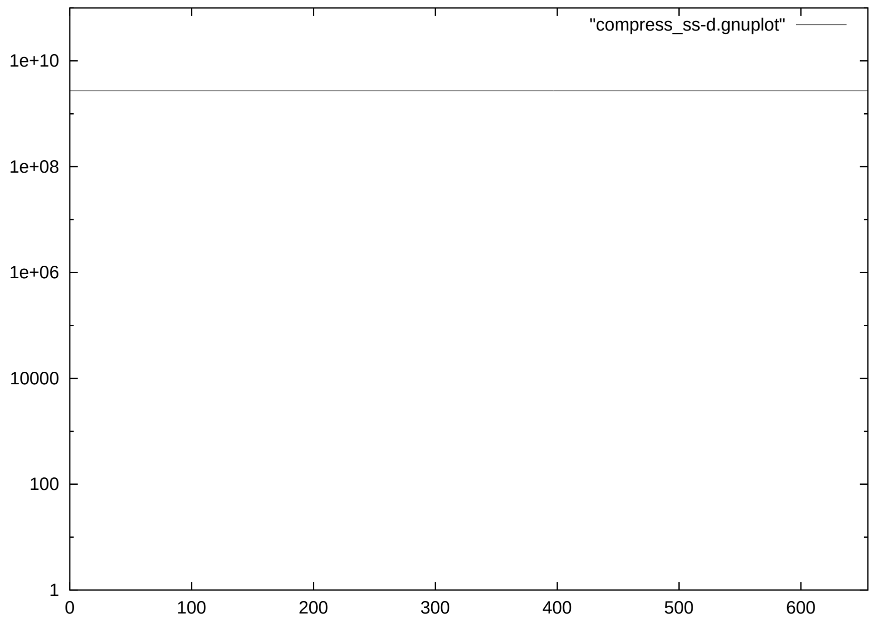

Figure B.1: MSE unique reference plot for *compress*. The x-axis gives the MSE-prescribed m (in hundreds of unique references); the y-axis gives the number of cold phase instructions.

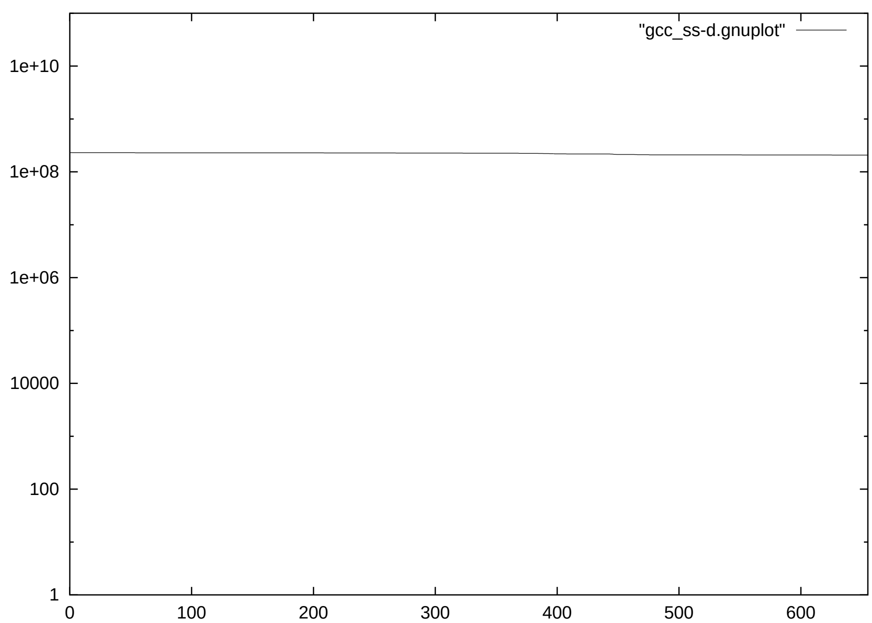

Figure B.2: MSE unique reference plot for *gcc*. The x-axis gives the MSE-prescribed m (in hundreds of unique references); the y-axis gives the number of cold phase instructions.

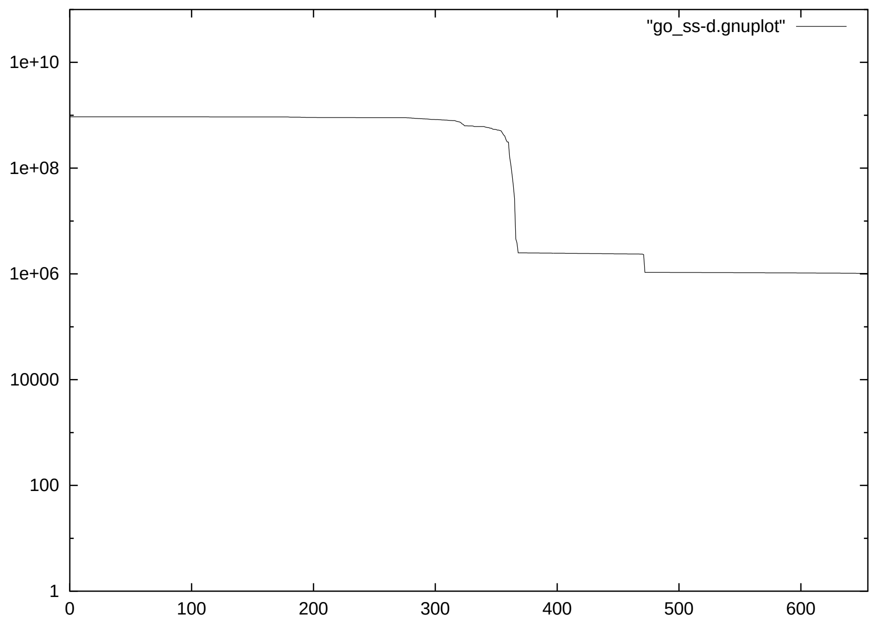

Figure B.3: MSE unique reference plot for *go*. The x-axis gives the MSE-prescribed m (in hundreds of unique references); the y-axis gives the number of cold phase instructions.

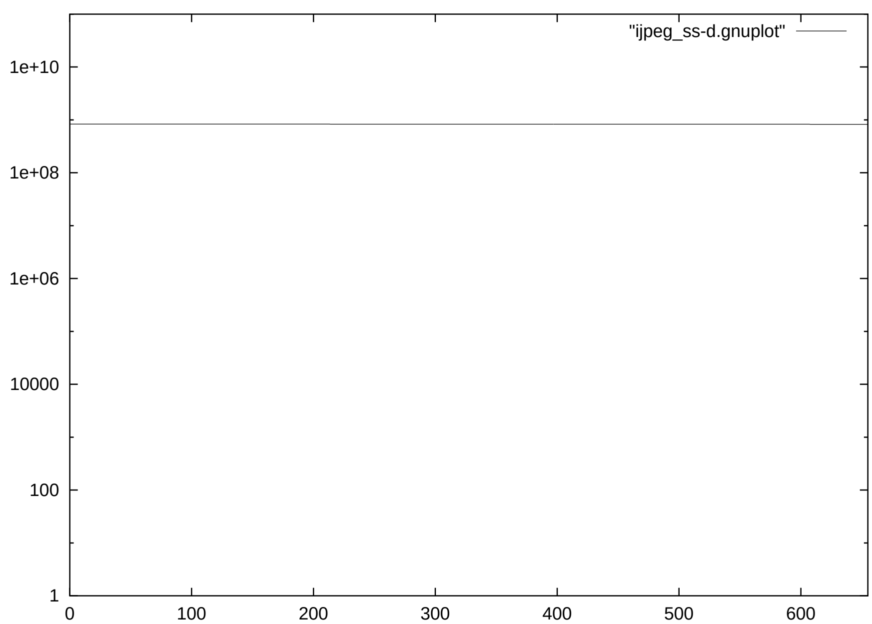

Figure B.4: MSE unique reference plot for *ijpeg*. The x-axis gives the MSE-prescribed m (in hundreds of unique references); the y-axis gives the number of cold phase instructions.

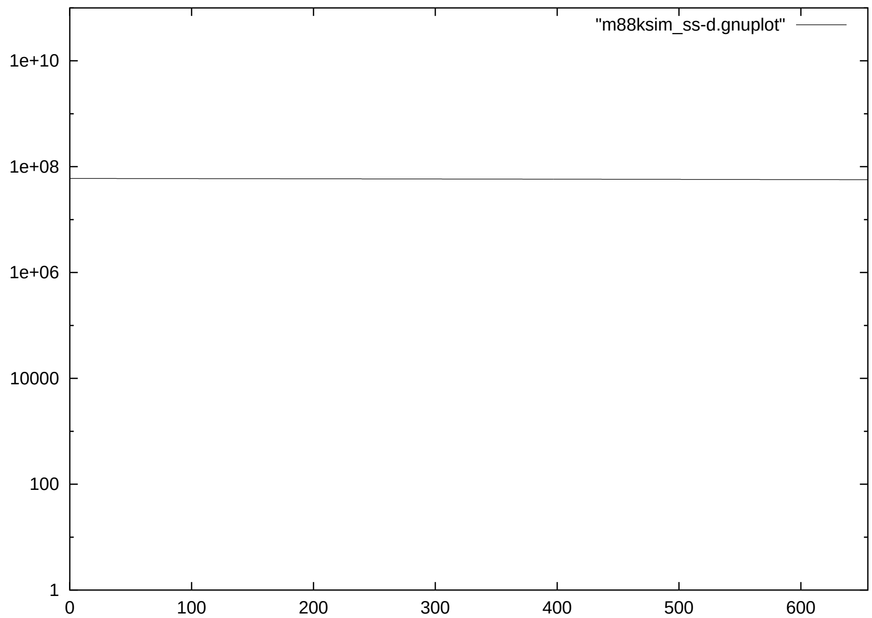

Figure B.5: MSE unique reference plot for *m88ksim*. The x-axis gives the MSE-prescribed m (in hundreds of unique references); the y-axis gives the number of cold phase instructions.

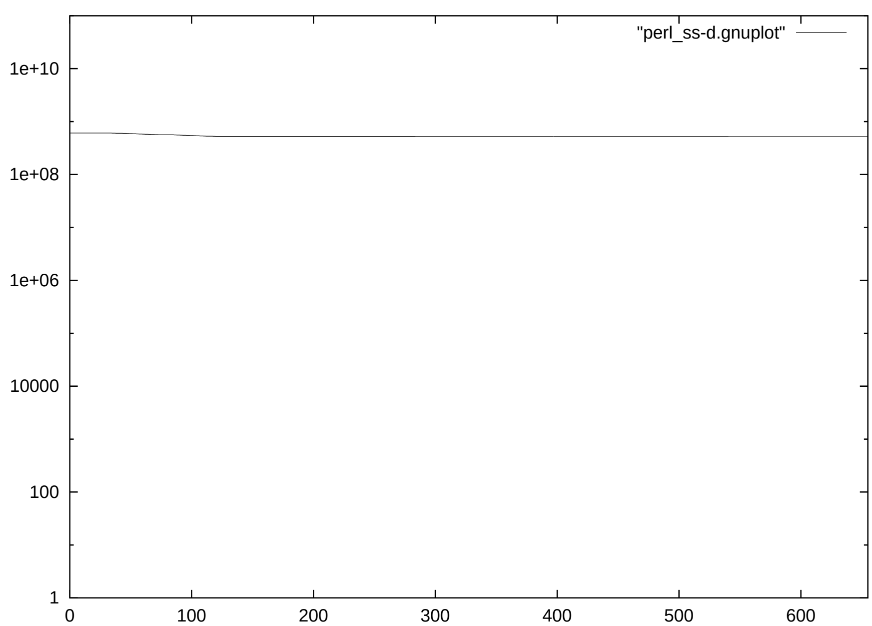

Figure B.6: MSE unique reference plot for *perl*. The x-axis gives the MSE-prescribed m (in hundreds of unique references); the y-axis gives the number of cold phase instructions.

Appendix C

MRRL Profiles

This appendix contains plots of the actual reuse latency profiles for several of the SPEC CPU2000 benchmarks used in this dissertation. Recall from Chapter 4 the bijective mapping of the discrete interval $[1, L]$ to the instructions of an L-instruction-long pre-cluster–cluster pair. Recall further, that reuse latency profiling works by first partitioning $[1, L]$ into n mutually-exclusive buckets whose union is the entire interval (*e.g.*, $[1, L] = \bigcup_{k=1}^{n} \text{bucket}_k$). Each bucket_i represents a subset $[a, b]$ of the interval where $a \geq 1$ and $b \leq L$. As the profiler proceeds, it counts the number of completed instructions, say C, between consecutive accesses to each unique reference address and increments the count of the bucket_i whose interval subset contains C. When the entire pre-cluster–cluster pair has been profiled, the profiler outputs the reuse latency data as histograms giving the population of each bucket_i for instructions, data, and branches.

This Appendix is organized into groups of three plots per page, where each page contains one benchmark. The top plot shows the reuse latency histogram for instructions; the middle plot shows the reuse latency histogram for data; and the bottom plot shows the reuse latency histogram for branches. The x-axis of each plot represents the interval $[1, L]$, and each bar corresponds to a bucket$_i$. The leftmost bar, bucket$_n$, corresponds to the interval subset $[a, L]$ and therefore counts the number of references that take the most instructions between consecutive accesses (*i.e.*, references with the longest reuse latencies). The rightmost bar on the other hand, bucket$_1$, corresponds to the interval subset $[1, b]$ and counts the number of references that take the least instructions between consecutive accesses (*i.e.*, references with the shortest reuse latencies). Notice on the logarithmically scaled y-axis, that bucket$_1$ is overwhelmingly the tallest, invariably. This indicates that a massive majority of memory reference addresses require very few instructions between successive accesses, precisely as would be expected in light of temporal locality [21].

The presence of most reuse latencies in bucket$_1$ justifies MRRL's initial insight and explains its ability to successfully speed up simulations relative to *fullwarmup*. As stated before, the MRRL insight reasons that if $N \times 100\%$ of reference addresses require only w_N instructions between consecutive accesses, then for large enough N, it is pointless to warm up references that occur more than w_N instructions before a sample cluster. The plots clearly show a very high reuse latency percentage in bucket$_1$. Hence, if $|\text{bucket}_i|$ is the count of references in bucket$_i$ and $N = \frac{\sum_{k=1}^{q} |bucket_k|}{\sum_{k=1}^{n} |bucket_k|}$, then very often, $q \ll n$. In other words, as one cumulatively amasses the population of bucket$_i$s, achieving $N \times 100\%$ of the population of all reuse latency measurements very often occurs in significantly fewer (q) than n buckets. Since $q \ll n$, if bucket$_q$ represents the interval subset $[a_q, b_q]$ and bucket$_n$ represents the interval subset $[a_n, L]$, then $b_q \ll a_n$. This means that the reuse latencies counted by bucket$_q$ have a short duration, such that warm up beginning at b_q instructions prior to the pre-cluster–cluster boundary will warm up significantly fewer instructions than warming up the entire pre-cluster phase. In short, it is preferable to have most of the histogram "mass" in the far right of each plot.

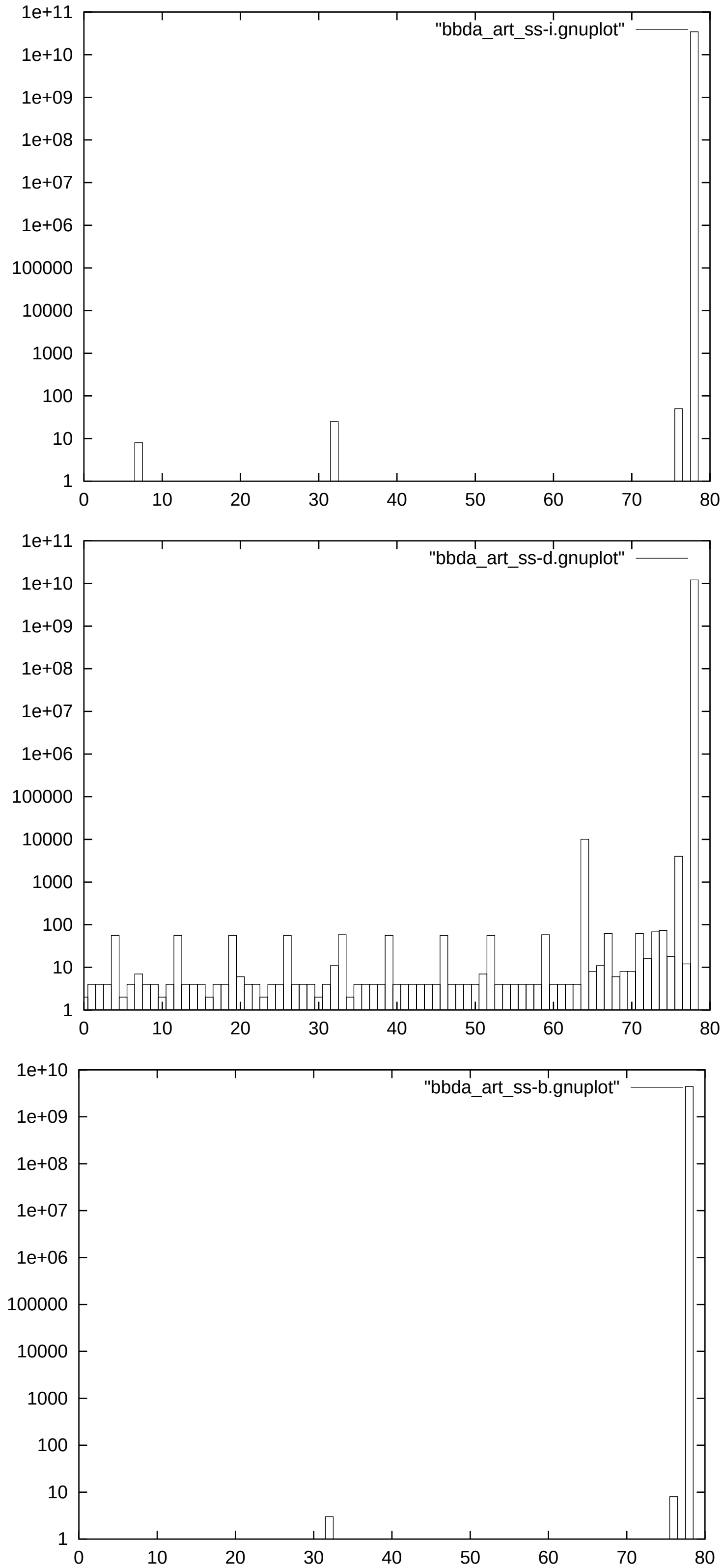

Figure C.1: MRRL plot for *art*. The x-axis corresponds to the bucket$_i$s; the y-axis gives the count of reuse latency measurements within each bucket.

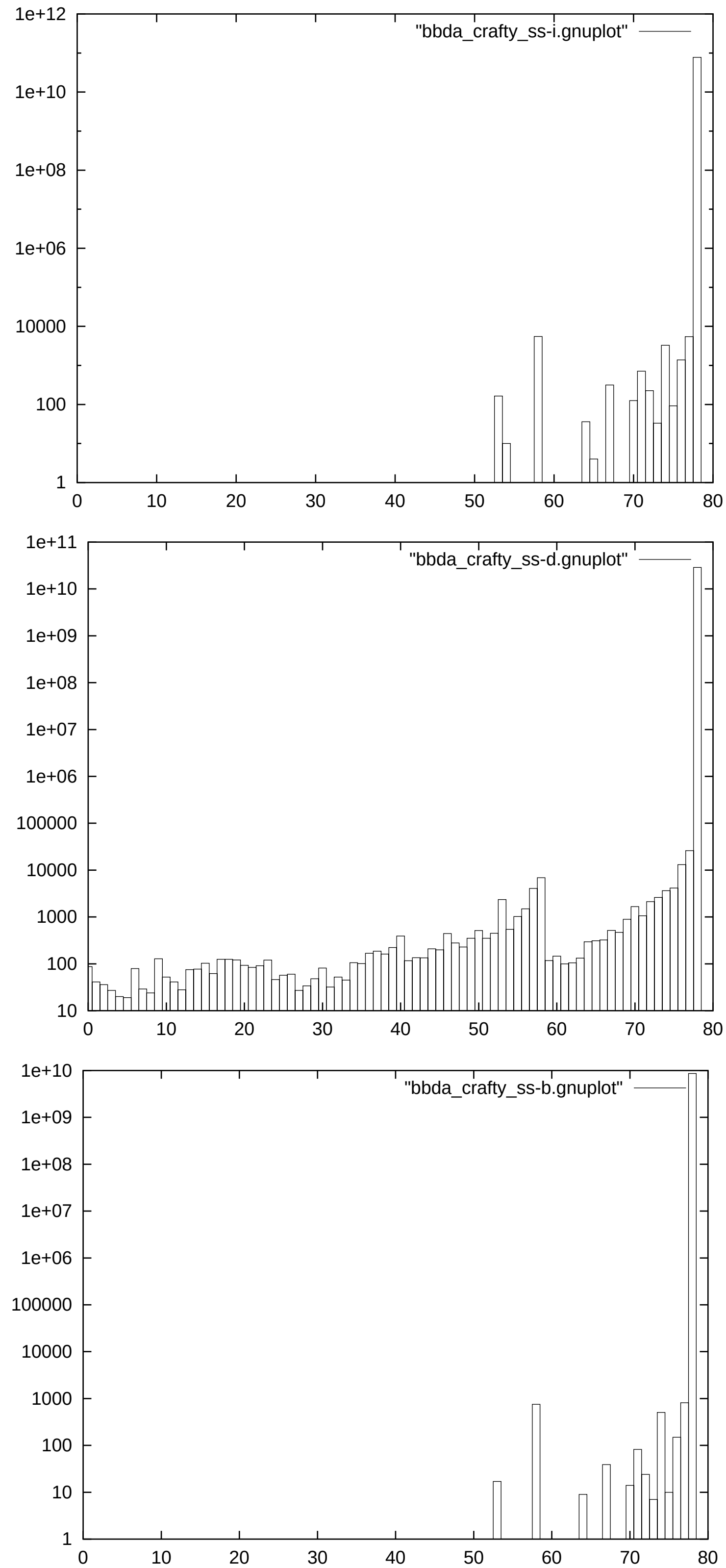

Figure C.2: MRRL plot for *crafty*. The x-axis corresponds to the bucket$_i$s; the y-axis gives the count of reuse latency measurements within each bucket.

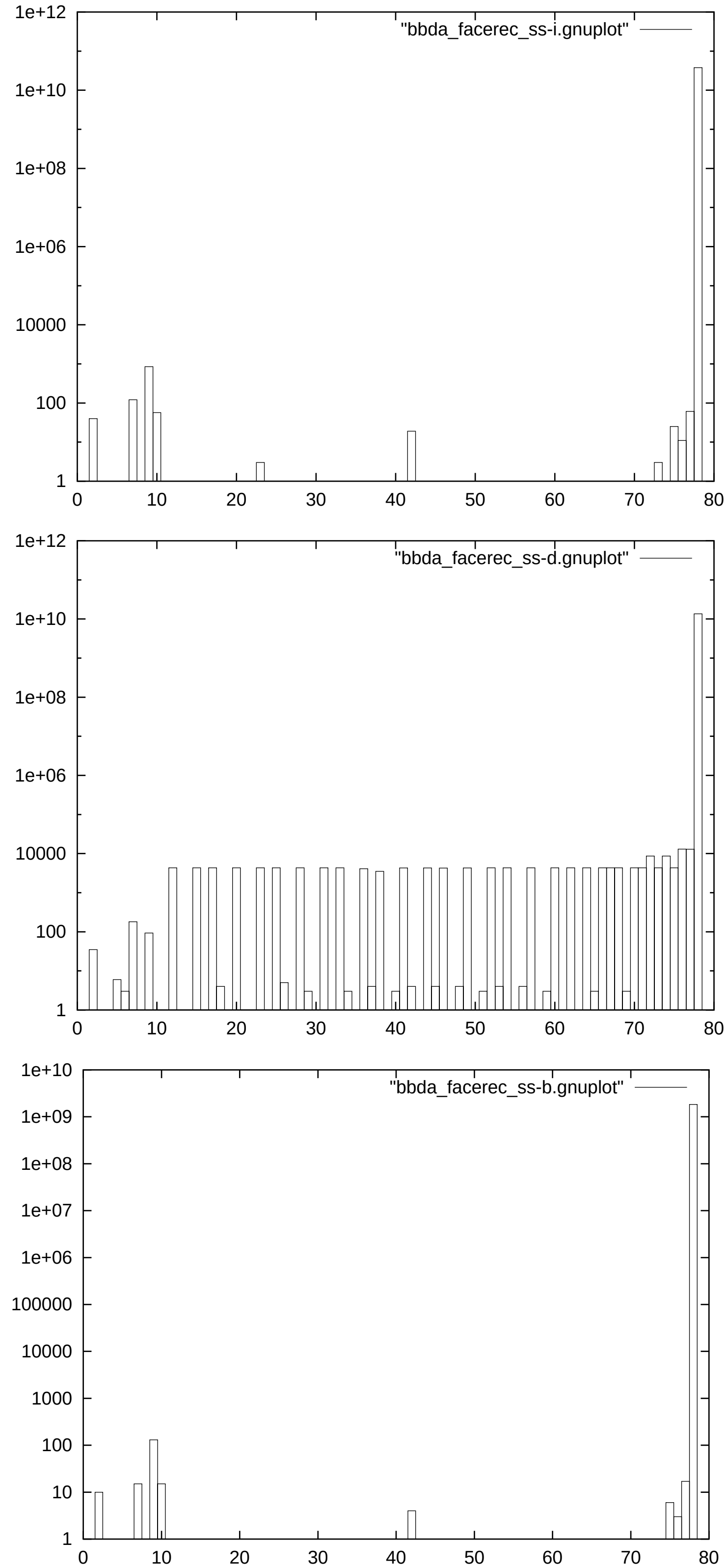

Figure C.3: MRRL plot for *facerec*. The x-axis corresponds to the bucket$_i$s; the y-axis gives the count of reuse latency measurements within each bucket.

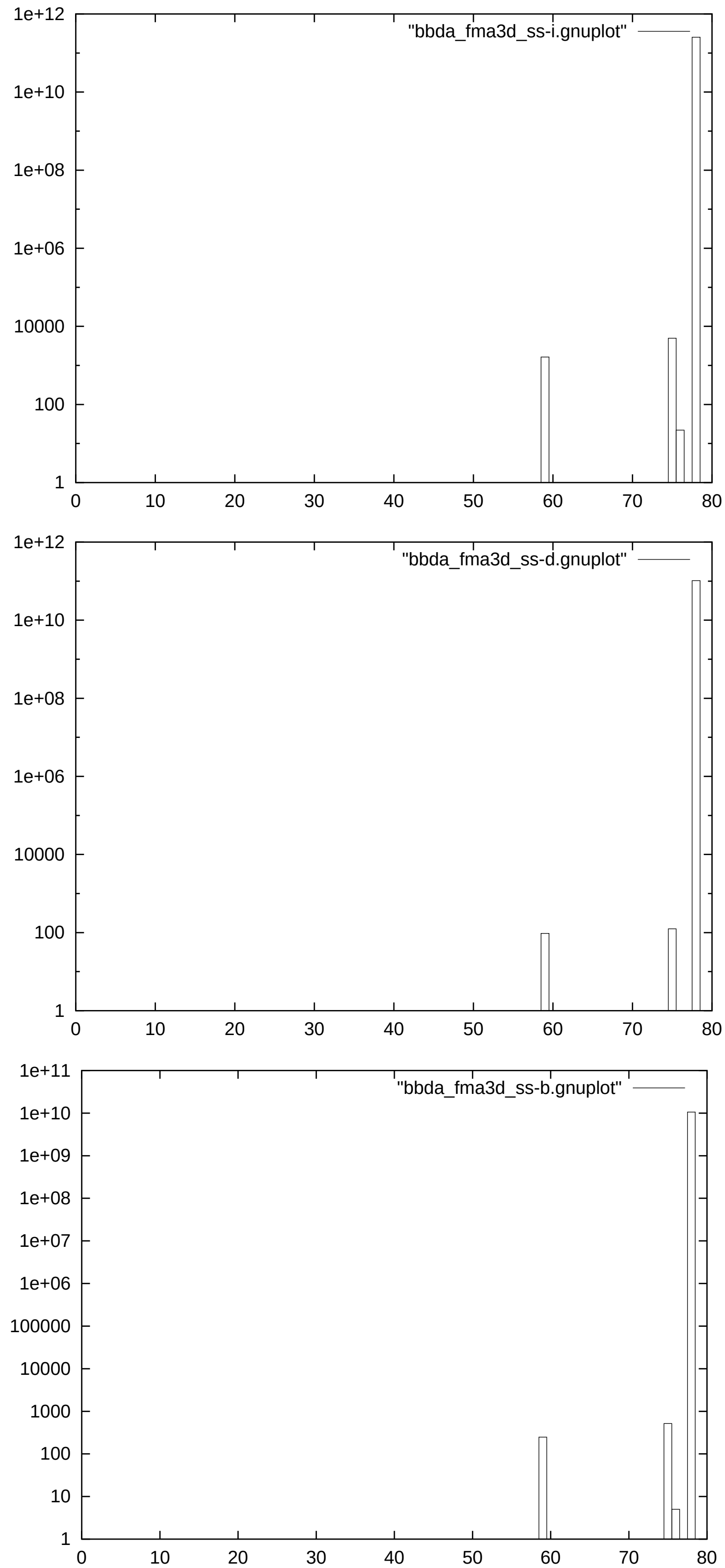

Figure C.4: MRRL plot for *fma3d*. The x-axis corresponds to the bucket$_i$s; the y-axis gives the count of reuse latency measurements within each bucket.

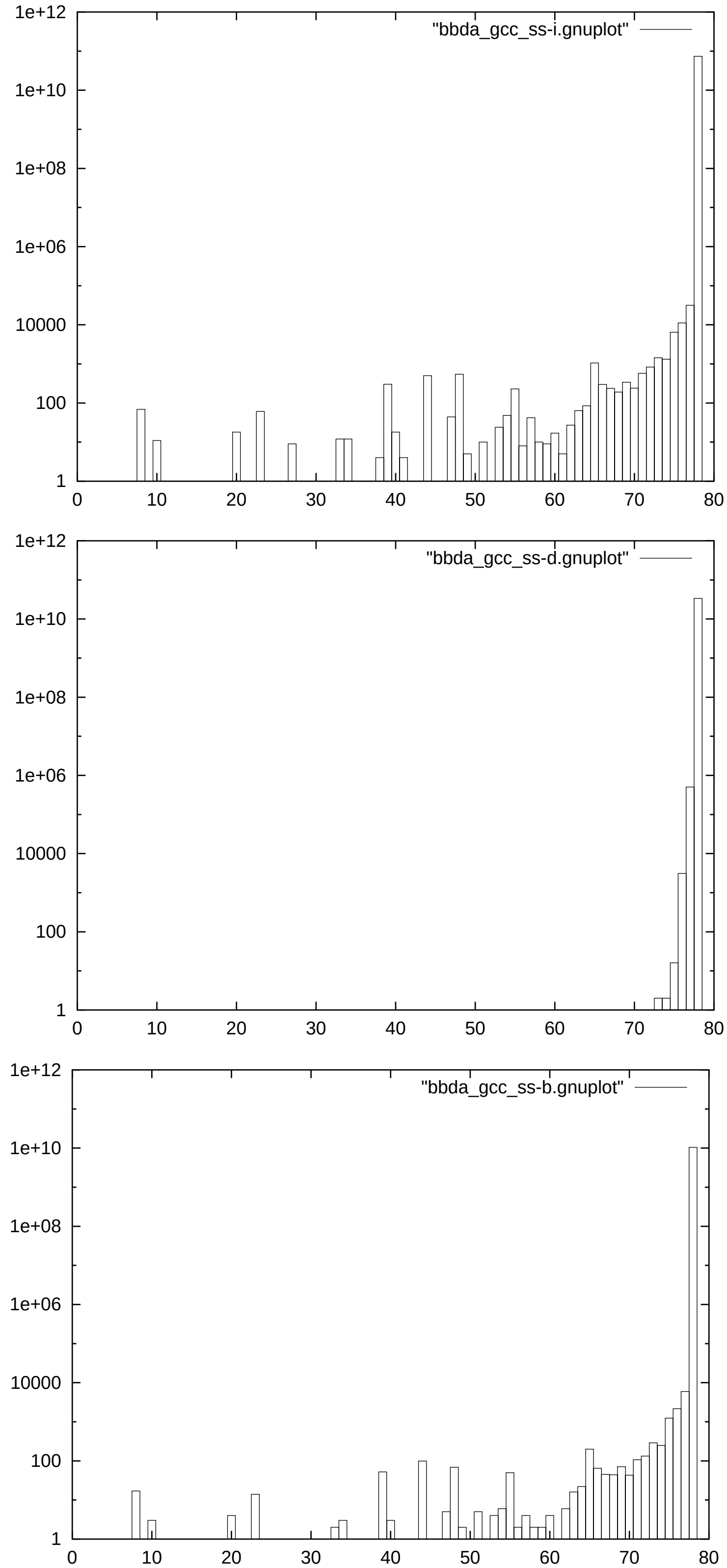

Figure C.5: MRRL plot for *gcc*. The x-axis corresponds to the bucket_is; the y-axis gives the count of reuse latency measurements within each bucket.

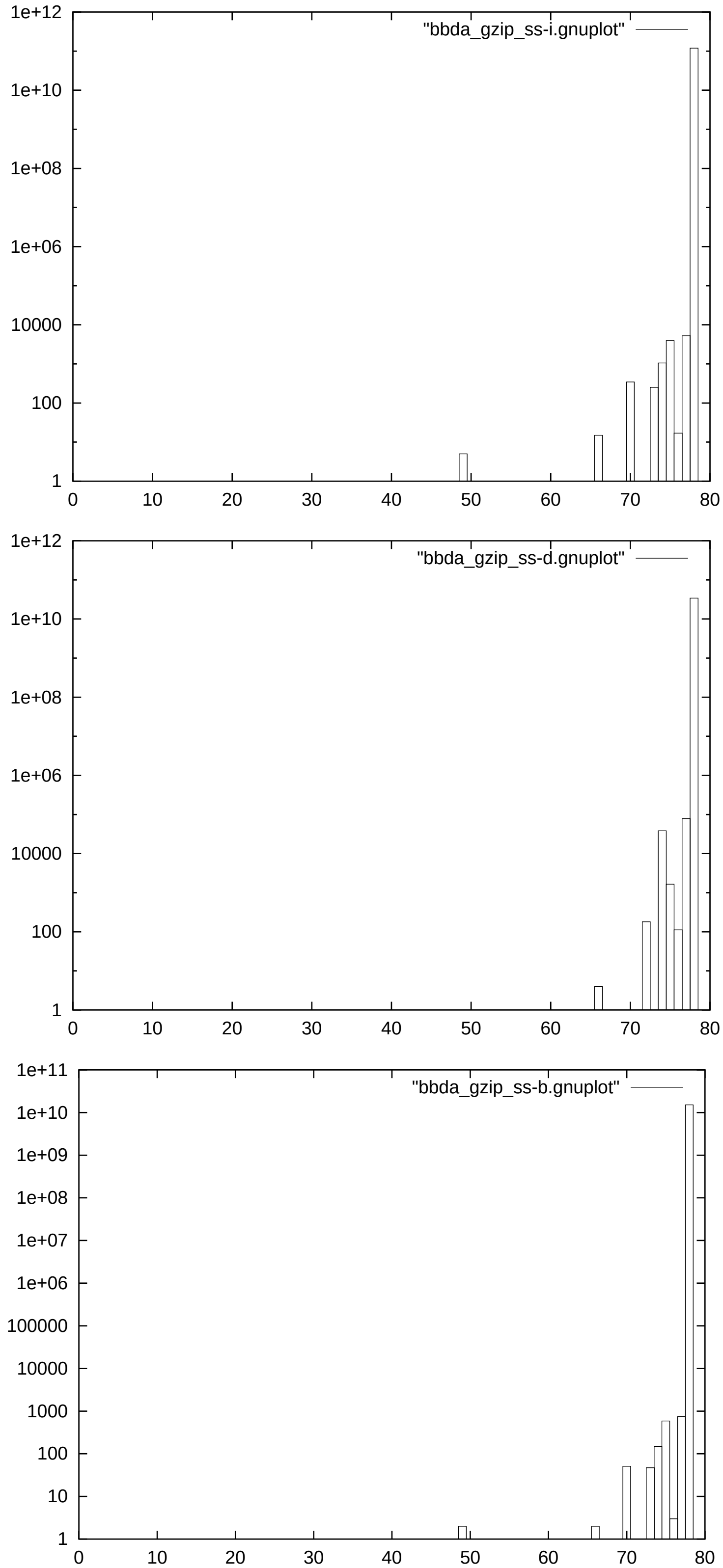

Figure C.6: MRRL plot for *gzip*. The x-axis corresponds to the bucket$_i$s; the y-axis gives the count of reuse latency measurements within each bucket.

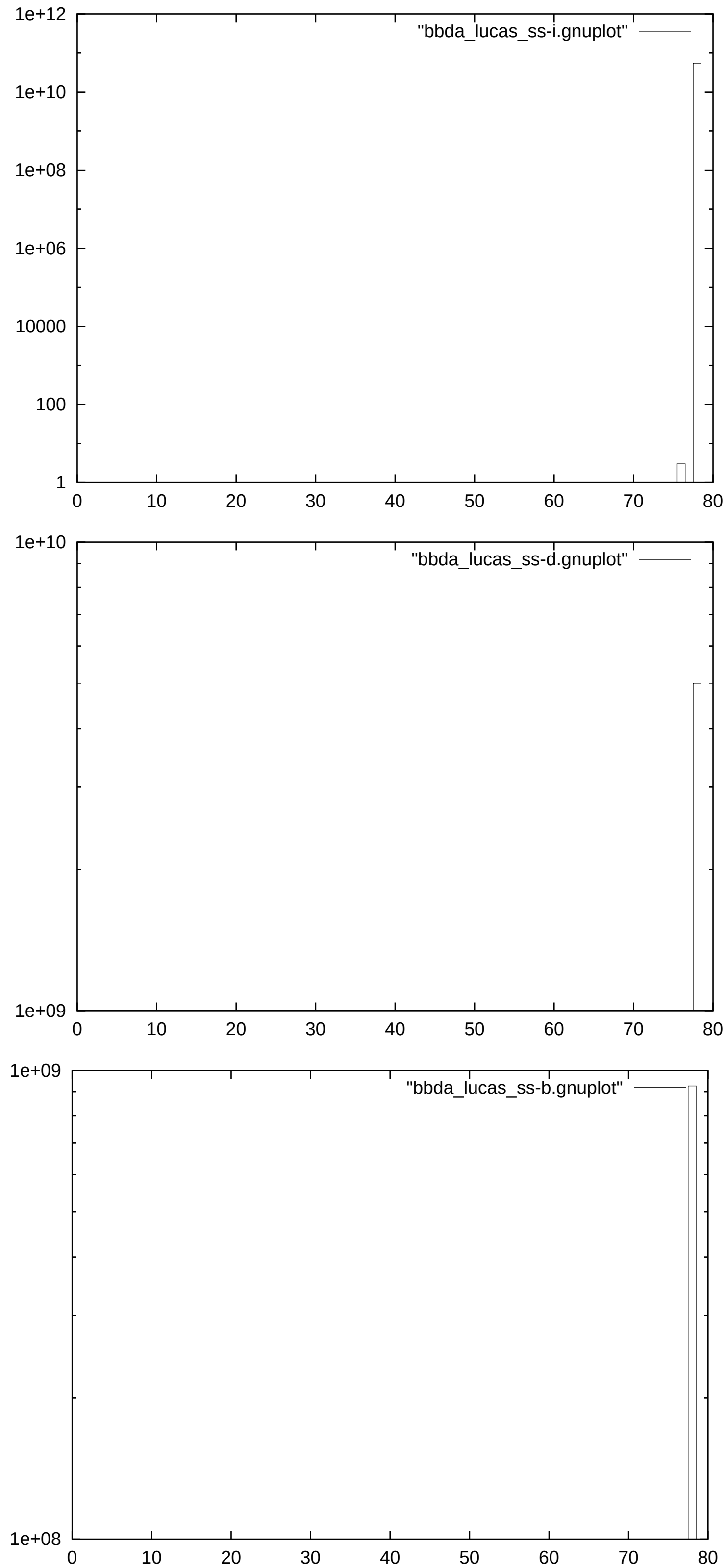

Figure C.7: MRRL plot for *lucas*. The x-axis corresponds to the bucket$_i$s; the y-axis gives the count of reuse latency measurements within each bucket.

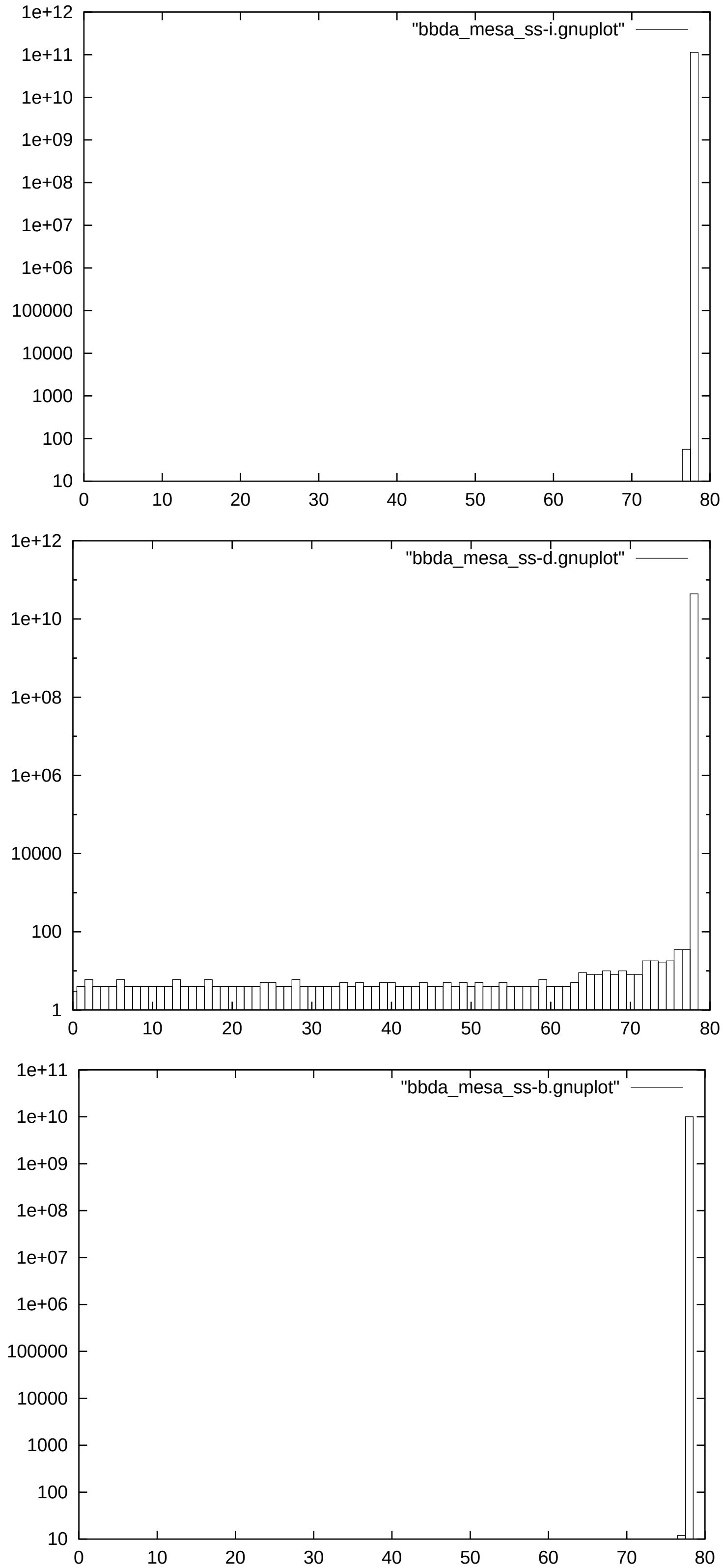

Figure C.8: MRRL plot for *mesa*. The x-axis corresponds to the bucket$_i$s; the y-axis gives the count of reuse latency measurements within each bucket.

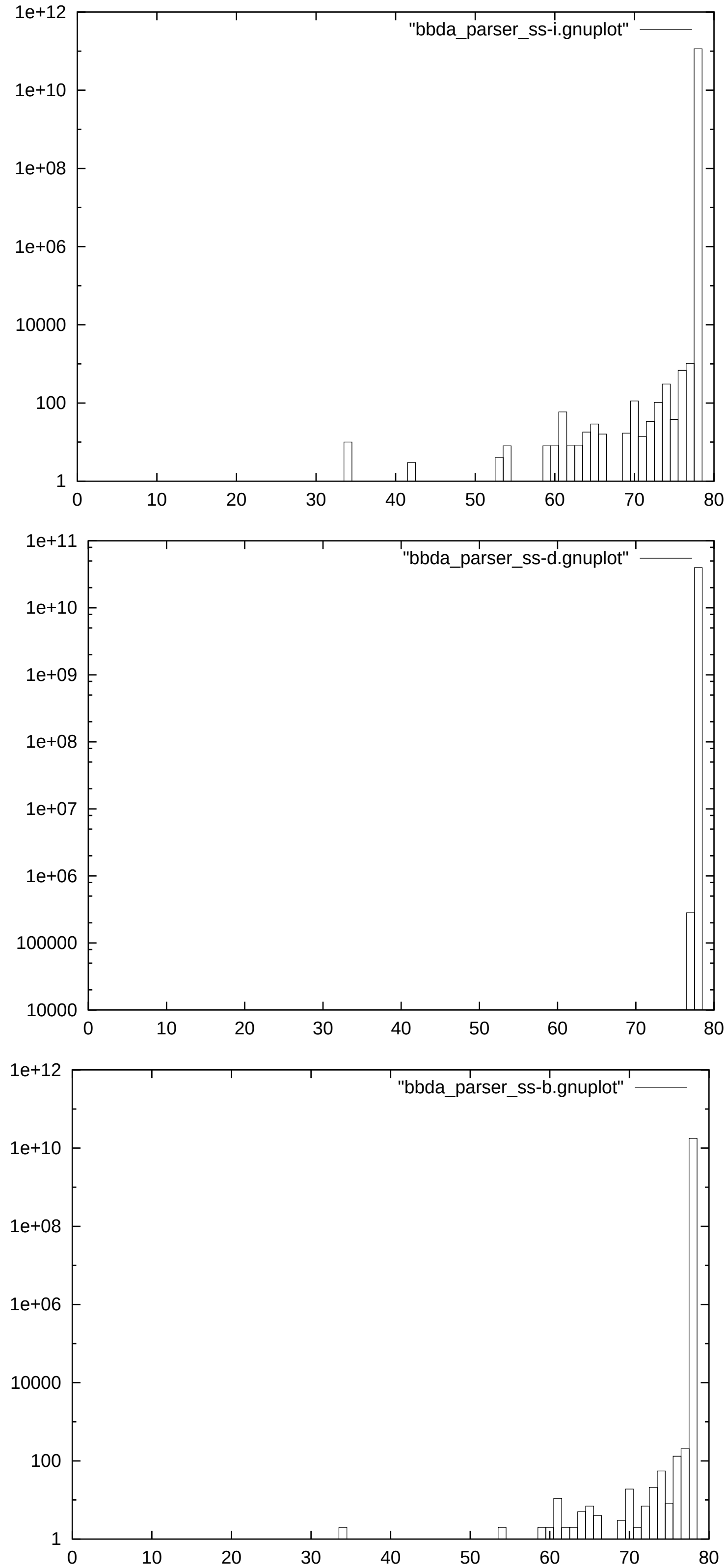

Figure C.9: MRRL plot for *parser*. The x-axis corresponds to the bucket$_i$s; the y-axis gives the count of reuse latency measurements within each bucket.

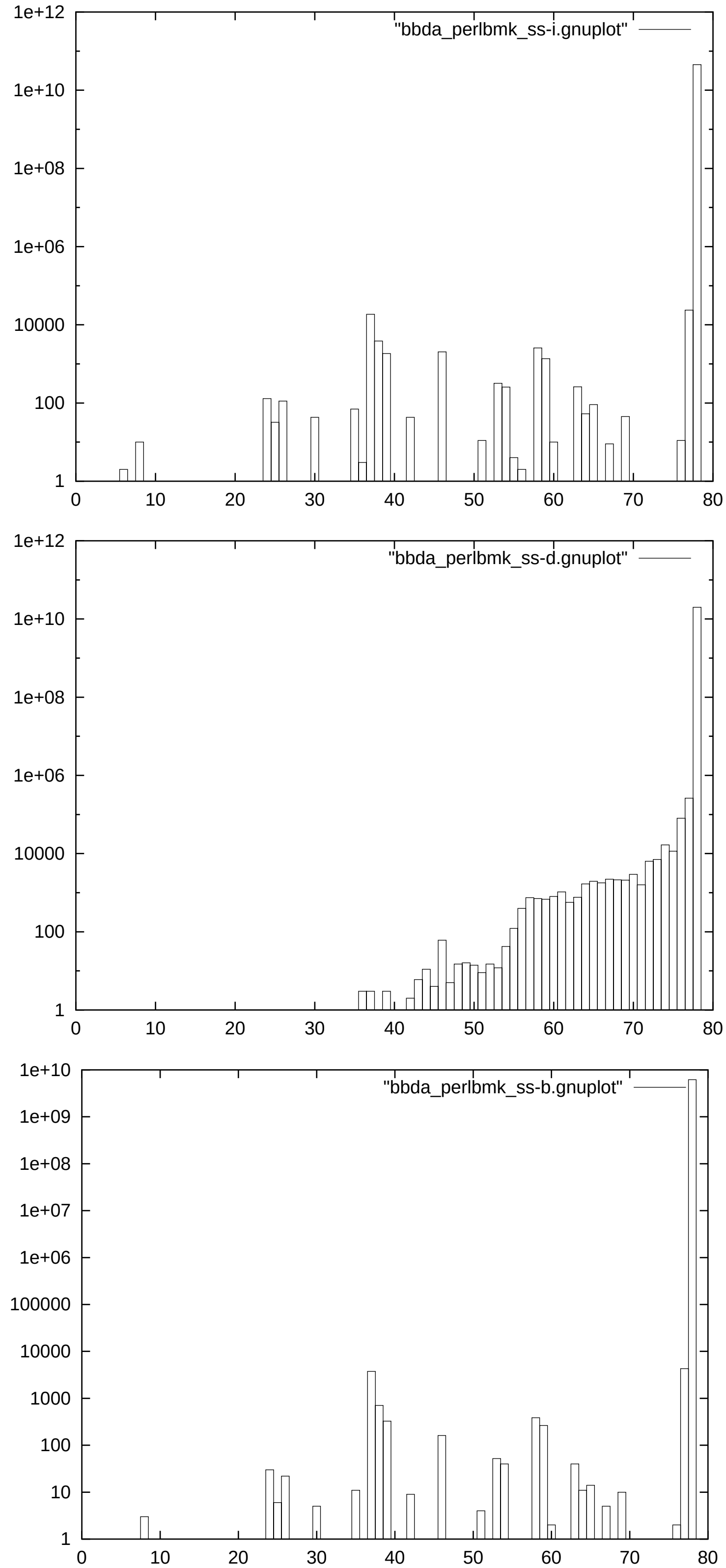

Figure C.10: MRRL plot for *perlbmk*. The x-axis corresponds to the bucket$_i$s; the y-axis gives the count of reuse latency measurements within each bucket.

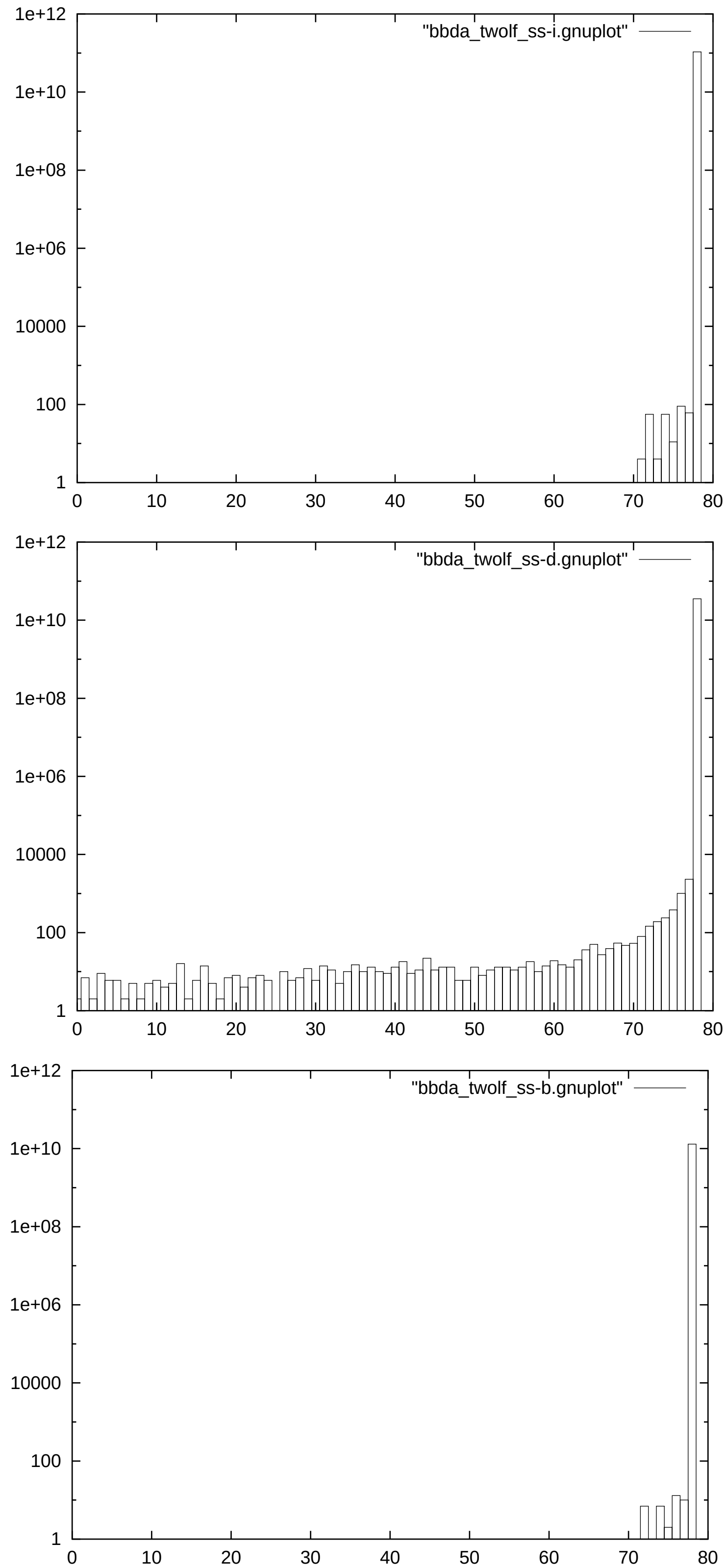

Figure C.11: MRRL plot for *twolf*. The x-axis corresponds to the bucket$_i$s; the y-axis gives the count of reuse latency measurements within each bucket.

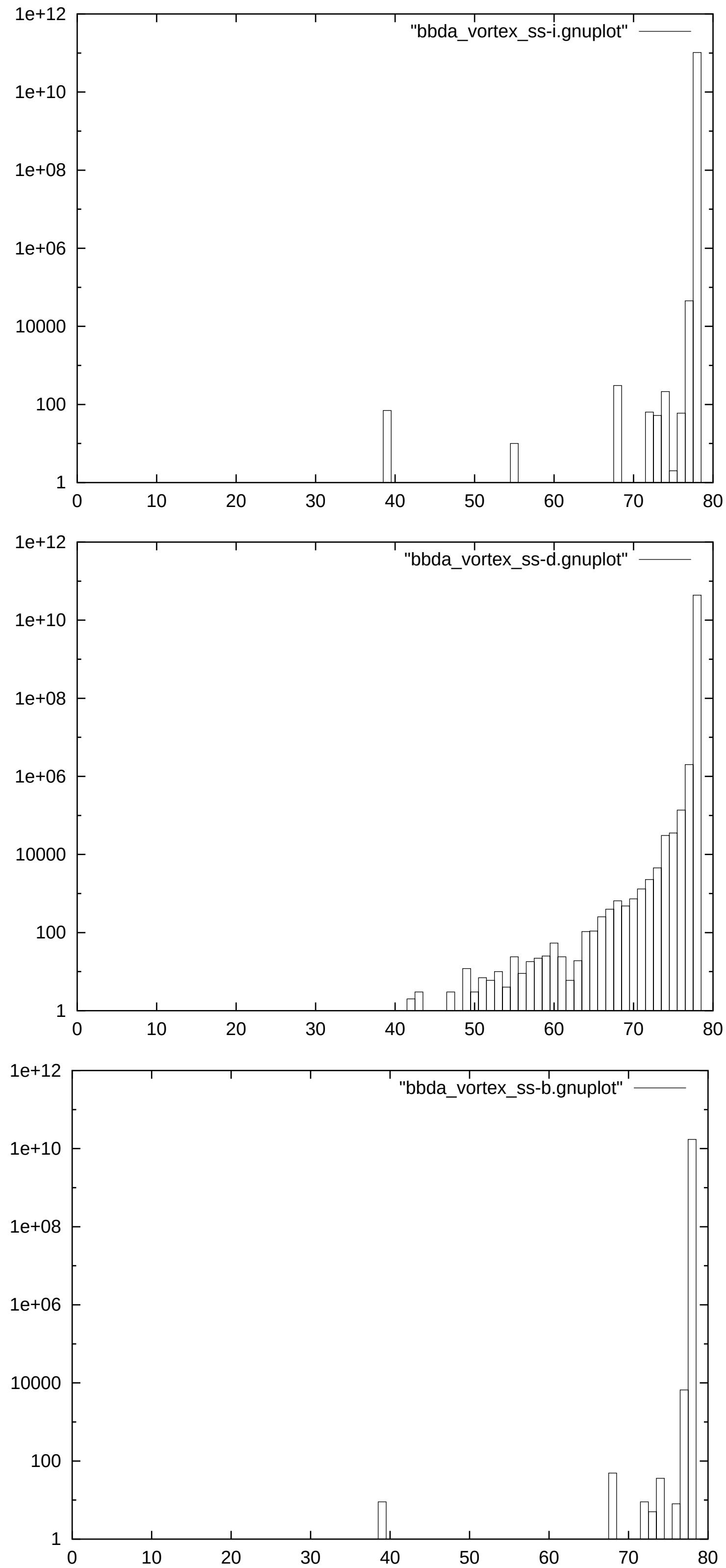

Figure C.12: MRRL plot for *vortex*. The x-axis corresponds to the bucket$_i$s; the y-axis gives the count of reuse latency measurements within each bucket.

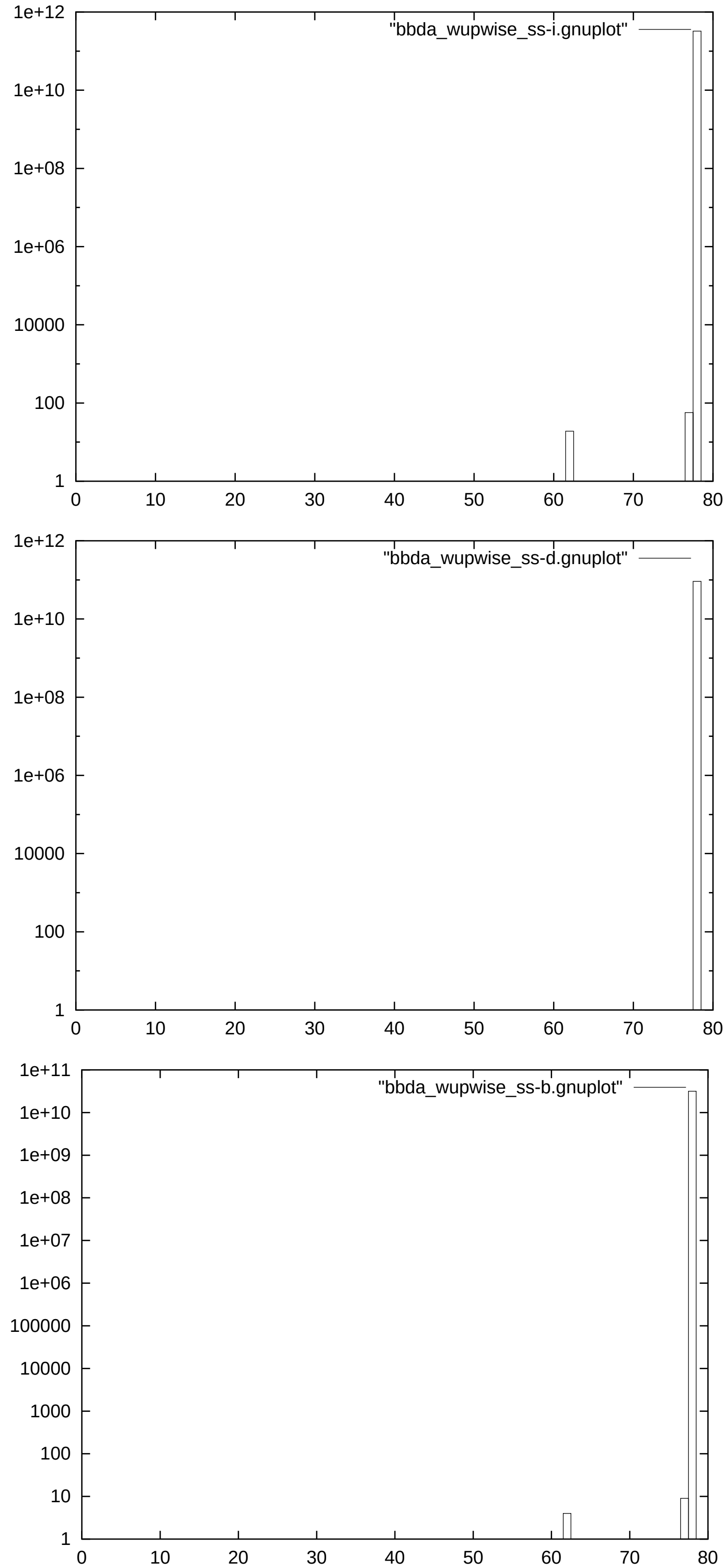

Figure C.13: MRRL plot for *wupwise*. The x-axis corresponds to the bucket$_i$s; the y-axis gives the count of reuse latency measurements within each bucket.