FIESTA: A Sample-Balanced Multi-Program Workload Methodology

Andrew Hilton, Neeraj Eswaran, and Amir Roth
Computer and Information Science Department, University of Pennsylvania
{adhilton, neeraj, amir}@cis.upenn.edu

Abstract

Workload construction methodologies for multi-program experiments are more complicated than those for single-program experiments. Fixed-workload methodologies pre-select samples from each program and use these in every experiment. They enable direct comparisons between experiments, but may also yield runs of which significant portions are spent executing only the slowest program(s). Variable-workload methodologies eliminate this load imbalance by using the multi-program run to define the workload, normalizing performance to the performance of the resulting individual program regions. However, they make direct comparisons difficult and tend to produce workloads that over-estimate throughput and speedup.

We propose a multi-program workload methodology called FIESTA which is based on the observation that there are two kinds of load imbalance. Sample imbalance is due to differences in standalone program running times. Schedule imbalance is due to asymmetric contention during multi-program execution. Sample imbalance is harmful because it dilutes multi-program behaviors. Schedule imbalance is a characteristic of concurrent execution that should be preserved and measured. Traditional fixed-workload methodologies admit both kinds of imbalance. Variable-workload methodologies eliminate both kinds of imbalance. FIESTA is a fixed-workload methodology that eliminates only sample imbalance. It does so by pre-selecting program regions for equal standalone running times rather than for equal instruction counts.

1. Introduction

Computer architecture research uses *multi-program workloads* to evaluate contention in shared caches and off-chip memory bandwidth in multi-core architectures, policies for assigning programs to cores in heterogeneous multi-core architectures [7, 8], thread scheduling and pipeline resource partitioning policies for multi-threaded architectures [1, 3, 4, 5, 10, 9, 12, 16, 18, 19, 20, 22], and job scheduling [15]. For these studies, multi-program workloads are more attractive than multi-threaded applications, because programs do not share instructions and data and tend to have a wide range of characteristics. This paper examines methodologies for constructing multi-program workloads for architecture experiments. We focus on methodologies that construct workloads from program samples rather than from complete programs. Architecture experiments often use slow simulation and sampling is an important technique for reducing experimental runtimes [11, 24].

Fixed-workload methodologies pre-select program samples and use them repeatedly in every experiment. A fixed-workload methodology may, for example, execute 100 million instructions from each program. Fixed-workload schemes align with the intuition that workloads are defined independently of the experiments that use them, support a clear and unambiguous interpretation of relative performance, and enable results from different experiments to be compared directly. However, they can also produce runs of which significant portions are spent executing only the slowest program(s). This characteristic, which is called *load imbalance*, is an experimental artifact. It is not present in real multi-program environments that process large numbers of programs and execute continuously. It creeps into experiments, which are finite. Load imbalance dilutes multi-program behaviors with single-program behaviors and causes the effects of concurrent execution to be underestimated.

Variable-workload methodologies eliminate experimental load imbalance by using the multi-program experiment itself to define the workload—*i.e.*, the workload is defined as the region from each program that executes during the multi-program run. An example variable workload methodology may execute all programs until each one has executed at least 100 million instructions. The drawback here is that each multi-program experiment may execute a different number of total instructions and/or different code regions from each program. This makes it impossible to directly

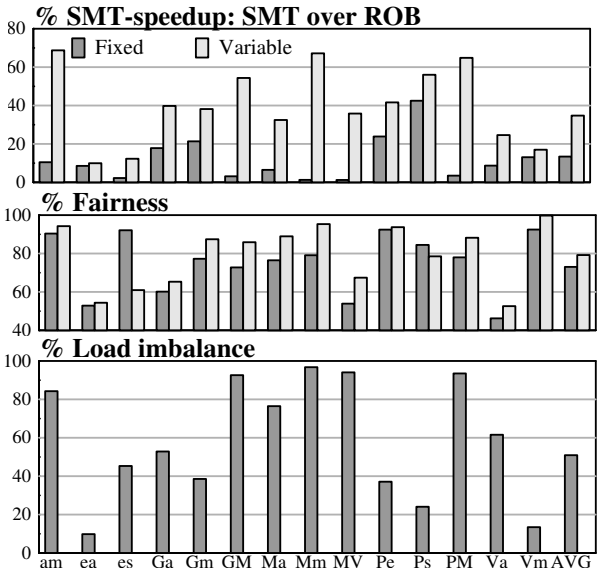


Figure 1: Fixed and Variable methodologies.

compare results from different experiments [14]. Normalized metrics like SMT-speedup enable only indirect comparisons. A more subtle problem is that variable-workload schemes can skew the workload in a direction that causes over-estimation of both throughput and SMT speedup. "Fairness" metrics [6, 10] attempt to account for this effect, but the relationship between SMT-speedup and fairness is not clear.

Figure 1 shows two multi-program experiments using eight of the SPEC2000 benchmarks. Each experiment pairs two programs on a 4-way issue SMT processor using the ICOUNT [19] policy (Section 3 details our methodology). In the fixed-workload experiment (Fixed), each sample executes 5 million instructions from each program. In the variable-workload experiment (Variable), each sample comprises a total 10 million instructions, although not necessarily 5 million instructions from each program. Both experiments execute 50 samples. The Variable experiment reports higher SMT-speedups (35% to 13% on average), somewhat higher fairness (79% to 73%), and much lower load imbalance measured as the percentage of total execution time during which only one thread is executing (0% to 51%). How do we interpret this data?

We propose *FIESTA*, a multi-program workload construction methodology which draws on the best features of traditional fixed- and variable- workload methodologies. The observation behind FIESTA is that fixed-workload multi-program experiments exhibit two different kinds of imbalance. *Sample imbalance* results from program samples of differing standalone running times and is present even when programs do not contend during concurrent execution. *Schedule imbalance* results from contention—in the caches and, in the case of SMT, in the pipeline—and from the architecture's response to it. Sample imbalance is problematic because it dilutes multi-program behavior. Schedule imbalance is a characteristic of concurrent multi-program execution, it should be preserved and measured. FIESTA is a fixed-workload methodology that eliminates sample imbalance, but does not attempt to reduce schedule imbalance. In FIESTA, corresponding samples from different programs have equal standalone execution times. FIESTA is an acronym for Fixed-Instruction with Equal STAndalone execution time. Continuing with the example, FIESTA would pre-select program samples that execute for 100 million *cycles* individually and then use these samples in every subsequent experiment. FIESTA retains the benefits of traditional fixed-workload methodologies. It supports an unambiguous interpretation of speedup and direct comparison of results from different experiments. FIESTA avoids the throughput and speedup over-estimation problems associated with variable-workload methodologies by not allowing contention to influence the workload. Table 1 summarizes the terms and concepts introduced and used in this paper.

FIESTA is well-suited for studying "policies"—*e.g.*, thread scheduling, resource partitioning—using experiments which share an underlying single-program execution model (Section 3.2). A *potential* drawback of FIESTA is architecture senstivity. Because FIESTA constructs workloads based on execution times, it guarantees sample balance only on the architecture on which the workloads were created. However, our experiments show that FIESTA workloads remain balanced even across architectures, making FIESTA a reasonable methodology for cross architecture studies (Section 3.3). Section 4 discusses other issues in multi-program workload construction and their relationship to FIESTA.

FIESTA is a simple and intuitive. Nevertheless, it is not used in any of the works we have examined, most of which use variable-workload schemes [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 18, 20, 22, 23]. We believe that this is because the distinction between sample imbalance and schedule imbalance has not been clearly articulated. As a result, the pitfalls of trying to eliminate schedule imbalance have not been understood. Articulating this distinction—and the pitfalls that come from ignoring it—is a major contribution of this paper. FIESTA follows directly from this distinction. Our hope is that FIESTA will replace the variable-workload methodologies which are commonly used today.

Multi-program workload. A set of program samples used in concurrent execution experiments.
Load imbalance. A situation in which a significant portion of a multi-program experiment is spent executing only the slowest program samples. Load imbalance dilutes multi-program behavior with single-program behavior and is an artifact of finite experiments.
Sample imbalance. Load imbalance that results from program samples with different standalone execution times.
Schedule imbalance. Load imbalance that results from asymmetric contention between program samples during concurrent execution.
Under(over)-sampling. A sampling pathology in which a program sample contains relatively less (more) of a behavior than the complete unsampled program.
Fixed workload methodology. A methodology that executes the same program samples in every multi-program experiment. Fixed workloads support unambiguous interpretation of speedup and direct comparisons across experiments, but they admit load imbalance.
Variable workload methodology. A methodology that executes different program samples in every multi-program experiment. Usually the multi-program experiment itself defines the workload. Variable workloads eliminate load imbalance by construction, but do not support direct comparison of results across experiments.
FIESTA (Fixed-Instruction with Equal STAndalone execution time). A fixed workload methodology that eliminates sample imbalance by choosing program samples with equal standalone execution times. FIESTA preserves schedule imbalance.

Table 1: Terms and concepts.

2. Multi-Program Workload Methodologies

Figure 2 illustrates all three methodologies using three programs (A, B, and C) two workloads (A/B and C/B) and two architectures (Arch1 and Arch2). We consider methodolgies that execute program samples rather than complete programs. Without excessive loss of generality, the figure—and our experiments later on—use periodic sampling. The lengths of the three programs correspond to their dynamic instruction counts—DIC_p is the dynamic instruction count of program p. To obtain N samples, we divide each program into N periods, each with DIC_p/N instructions. For readability, the example uses 3 samples rather than the 50 used in our experiments. Each period i is divided into two parts. The sample contains S_{pi} instructions. The unsampled (*i.e.*, "fastforwarding") region contains F_{pi} instructions, where F_{pi} is computed as $DIC_p/N–S_{pi}$.

2.1. Fixed Workload Methodologies

In traditional fixed-workload methodologies, corresponding samples from each program contain equal instruction counts [1]. S_{pi} is constant across all programs and potentially all sampling periods. In the Fixed experiment in Figure 1, S_{pi} is 5 million instructions for all programs p and samples i. The fixed-workload methodology in Figure 2a selects 5 million instructions per sample for every program regardless of which architecture it executes on or which program it is paired with.

Drawback: load imbalance. Given inherent differences between programs, equal instruction counts may correspond to vastly different execution times. As a result, the multi-program run may spend a significant amount of time executing fewer than the maximum number of programs. This artifact is called *load imbalance*.

High load imbalance dilutes the effects of concurrent execution. One metric that measures the performance impact of concurrent execution is SMT-speedup [14]. SMT-speedup is defined as $(\sum_{i=progs} T_i^{SP}/T^{MP})$, where T_i^{SP} is the single-program (*i.e.*, standalone) execution time of program i and T^{MP} is the multi-program execution time. SMT-speedup is greater than 1—or positive if speedup is reported as a percentage rather than a factor or multiple—if the program samples execute faster concurrently than they do sequentially. Consider two cases in which programs A and B do not contend, and do not lose performance when executed concurrently. If A and B both execute for 1 million cycles each when executed alone, then the multi-program run also takes 1 million cycles. SMT-speedup is 100% and load imbalance is 0%. If A executes for 1 million cycles and B for 10 million, multi-program execution takes 10 million cycles. Here, SMT-speedup is only 10% and load imbalance is 90%. If only SMT-speedup is reported, we may infer high contention, where in fact none exists. If load imbalance is reported as well, then no significant conclusion can be drawn.

2.2. Variable Workload Methodologies

Variable workload methodologies eliminate load imbalance by using the multi-program experiment itself to define the workload. Figure 2b shows an example. Here, each multi-program sample contains a total of 10 million instructions although not necessarily 5 million instructions from each program. There are other ways of defining variable-workload samples—a sample could correspond to 10 million execution cycles, it could last until at least one program has excuted 5 million instructions, or until every program has executed at least 5 million instructions—but the overall effects are the same. Different samples of the same application may be selected depending on which application it is paired with—application B has fewer instructions in each sample when paired with "fast" application A, and

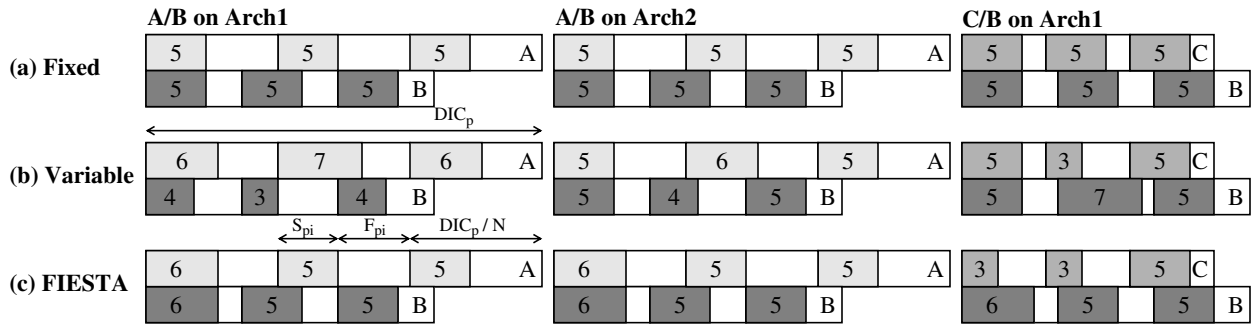


Figure 2: Multi-program workload construction methodologies. Example shows three programs A, B, and C whose horizontal length corresponds to their instruction counts. We use three methodologies to form two workloads: A/B and B/C. Each multi-program workload consists of three samples from each program. A fixed-workload methodology uses a fixed 5M instructions per program per sample. A variable-workload methodology uses a total of 10M instructions per sample, but the sample composition can change based on the specific pairing and the underlying architecture. FIESTA chooses samples such that correpsonding samples from different programs have the same standalone execution times. Samples are not necessarily the same across programs, but for a given program they are constant across pairings and architectures.

more when paired with "slow" application C. More importantly, the same application pair may yield different samples when executed on different architectures.

Drawback: incomparable experiments. When each experiment defines its own workload, reslts from different experiments are not directly comparable. Previous work has acknowledged this shortcoming [14], and argued that results from different experiments should be compared using indirect metrics that normalize the performance of the multi-program run by the standalone performance of the constituent program regions *post fact*. SMT-speedup is one such metric. SMT-speedup correctly reports the performance effects of concurrent execution *for a given experiment*. However, it does not follow that SMT-speedups computed on different workloads are comparable to each other.

Drawback: optimistic throughput and SMT-speedup results. A more subtle and serious problem is that the multi-program run may skew the workload in a way that leads to overly-optimistic throughput and speedup results. It is easy to see how a variable-workload methodology can result in over-estimation of throughput—it will naturally under-sample slow programs relative to faster ones. It can result in an over-estimation of SMT-speedup as a result of asymmetric contention. When two programs contend, they may not experience that contention—in terms of degraded performance—equally. This could be due to their *a priori* performance, the particular form of contention, or explicit decisions by the hardware (*e.g.*, the thread selection policy). When two programs contend asymmetrically, a variable-workload methodology will under-sample the program whose contention-induced slowdown is higher.

Asymmetric performance degradation is captured by "fairness." Gabor *et al.* [6] define fairness as $(\min_{i,j=progs}\ (T_i^{SP}/T_i^{MP})/(T_j^{SP}/T_j^{MP}))$. Fairness is 1—or 100%—if concurrent execution degrades the performance of all programs by the same relative amount. In a variable-workload methodology, low fairness indicates that either the experiment or a hardware policy (or both) is skewing speedup upwards. However, there is not a strong relationship between traditional fairness and speedup metrics that allows the "correct" speedup to be computed. This is a good time to point out that high load imbalance inflates traditional fairness. At high load imbalance, the performance degradation of the program that runs alone at the end of the sample approaches 0, leaving the reported fairness to be that of the shorter running program. If the shorter running program is the one that degrades unfairly, that fact is reported correctly. However, if the longer running program degrades unfairly, then that fact is hidden by the imbalance.

2.3. FIESTA

The observation that motivates FIESTA is that traditional fixed-workload methodologies experience two forms of imbalance. *Sample imbalance* is the result of program samples that have different standalone execution times. Sample imbalance may exist in a multi-program run even if programs do not contend or if they contend in a symmetric or fair way. *Schedule imbalance* results from asymmetric or unfair contention, *i.e.*, one program experiences contention as a larger slowdown than another. Sample imbalance is an experimental arti-

	Unsampled					Fixed		FIESTA		Variable	
Name	IPC	BMPK	LdLat	D$MLP	L2MLP	IPC	%Insn	IPC	%Insn	IPC	%Insn
Perl(P)	1.61	9	4	1.4	2.1	1.58	0.7	1.66	1.2	1.66 (1.66-1.66)	0.9 (0.6-1.3)
Gcc(G)	1.56	7	5	1.8	3.2	1.50	5.8	1.77	10.3	1.67 (1.57-1.78)	6.9 (4.4-10.1)
Mcf(M)	0.10	9	128	5.6	5.3	0.10	2.8	0.19	0.5	0.17 (0.16-0.19)	0.9 (0.3-1.9)
Vortex(V)	2.62	1	4	1.3	1.9	2.62	1.5	2.65	4.0	2.63 (2.63-2.64)	2.0 (1.3-2.6)
Equake(e)	1.01	<1	32	3.0	2.2	0.94	1.1	0.95	1.1	1.04 (0.95-1.18)	0.8 (0.4-1.5)
Art(a)	0.47	<1	298	33.8	21.2	0.47	6.9	0.46	3.2	0.46 (0.46-0.47)	4.5 (1.7-10.2)
Swim(s)	1.62	<1	58	12.5	8.5	1.65	2.9	1.70	4.9	1.70 (1.64-1.73)	3.4 (2.6-5.0)
Mesa(m)	3.15	<1	5	44.3	44.5	3.24	0.4	3.27	1.2	3.26 (3.23-3.30)	0.5 (0.4-0.7)

Table 2: Benchmark characterization. We characterize unsampled execution IPC, branch mis-predictions per 1K instructions, and D$ and L2 MLP. Workload samples are characterized using IPC and sampling rate (% of total dynamic instructions in the sample). Variable produces a range of samples for each program.

fact that does not exist in real, continuous multi-program environments. Schedule imbalance is a characteristic of concurrent execution.

We argue that workload-construction methodologies should attempt to eliminate only artificial sample imbalance; they should not try to eliminate schedule imbalance which is a real effect that should be preserved and measured. This is important because sample imbalance is easy to eliminate in a fixed-workload methodology by choosing program samples that have equal standalone running times rather than equal instruction counts. In contrast, only variable-workload schemes can eliminate schedule imbalance.

FIESTA is a fixed-workload methodology that eliminates sample imbalance, but does not attempt to eliminate schedule imbalance. FIESTA chooses application samples by first running each application alone. However, rather than specifying the number of instructions each sample should contain, S, FIESTA specifies the number of cycles C for which each sample should execute. During the standalone run, FIESTA records how many instructions (S_{pi}) each C-cycle sample corresponds to. These regions comprise the program's sample and they are used in all subsequent multi-program experiments. Figure 2c shows a FIESTA workload. As in a variable-workload methodology, the sample of one program need not contain the same number of instructions as the sample of a second program. However, as in a fixed-workload methodology, the sample of a given program is constant regardless of which architecture it runs on and which other program it is paired with. In Figure 2c, application B executes the same instruction region in each sample whether it is paired with A or C. Likewise, A and B execute the same instruction regions whether they are executing on Arch1 or Arch2—in these environments, workload composition parallels standalone program throughput.

As a fixed-workload methodology, FIESTA supports direct comparisons between experiments and it does not allow the multi-program experiment to influence the workload composition and, as a result, skew throughput and speedup. As such, FIESTA workloads correspond to continuous multi-program environments that are also "fair".

3. Experimental Evaluation

We evaluate FIESTA in the context of simultaneous multithreading (SMT). SMT leads to more contention between programs and highlights differences between methodologies.

Simulation tools. Our cycle-level simulator models a 4-way issue, dynamically scheduled superscalar processor with a 17-stage pipeline, 256-entry reorder buffer, 64-entry issue queue, and up to four threads. We model 64Kbyte, 8-way set-associative instruction and data caches, a 4Mbyte, 16-way set-associative, 15-cycle access L2 backed by eight 8-entry stream buffers, and 400 cycle main memory. It supports Runahead execution and Runahead Threads (RaT) [13]. We use a "capped" partitioning policy for the issue queue, miss-status holding registers (MSHRs), and stream buffers [12]. For a resource capacity C and a number of active threads T, a thread is guaranteed a minimum allocation of $C/2T$ entries and cannot acquire more than $C/2+C/2T$ entries. This policy produces the highest throughput.

Workload Methodologies. Each methodology chooses 50 samples from each program. Each FIESTA sample represents 5 million cycles of standalone execution. A *Fixed* sample is 5 million instructions. A *Variable* sample contains 10 million total instructions for a two-thread workload or 20 million total instructions for a four-thread workload.

Benchmarks. We use eight SPEC2000 benchmarks: floating-point programs *art*, *equake*, *mesa*, and *swim* and integer programs *gcc*, *mcf*, *perl*, and *vortex*. Each benchmark is referred to by a the first letter of its name, integer programs by capital letters (G, M, P, V) and

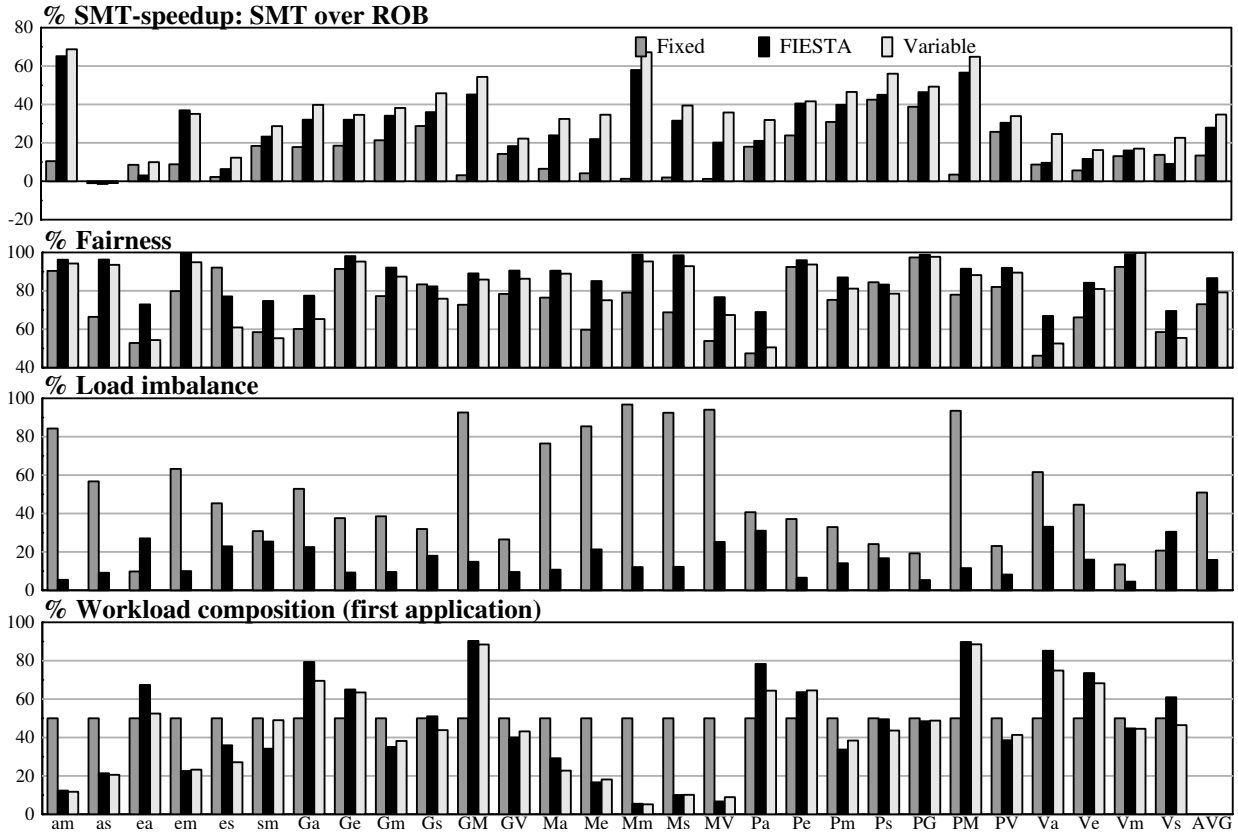


Figure 3: Two-thread experiments: SMT-speedup, load imbalance, fairness, and workload composition.

floating-point programs by lower case letters (*a*, *e*, *m*, *s*). We run two- and four- thread experiments for all combinations that use mixtures of different programs. Workloads are named by the concatenation of the one-letter program abbreviations, *e.g.*, *Ge* is the *gcc*/*equake* workload. There are 28 two-thread and 70 four-thread workloads. Averages shown are over all workloads.

Table 2 characterizes the unsampled execution of the benchmarks using IPC, branch mis-predictions per 1K instructions (BMPK), average load latency (LdLat), D$ miss level parallelism (D$MLP) and L2 miss level parallelism (L2MLP). We chose the benchmarks to represent four different behaviors: high-ILP (*vortex*, *mesa*), branch limited (*gcc*, *perl*), memory latency limited (*mcf*, *equake*), and memory bandwidth limited (*art*, *swim*).

The table also characterizes the samples chosen by each methodology using IPC and instruction sample rate (%Insn). Variable methodology results are shown as an average and a range in parentheses, because Variable chooses as many samples of each program as there are multi-program experiments. Fixed has low IPC sampling error because it samples uniformly on an instruction basis. FIESTA has a larger error because it is time-based—from an instruction-standpoint, it naturally under-samples slow regions and over samples faster ones. FIESTA almost always over-reports IPC and Section 4 explains how to address this problem. Variable has large average and maximum errors because multi-program runs under-sample *both* slow programs *and* programs that experience "unfair" slow-downs.

3.1. Comparing Workload Methodologies

Two-thread results. Figure 3 shows results for two-thread workloads: SMT-speedup, fairness, load imbalance, and workload composition—percentage of total instructions committed by the first program in the pair. FIESTA reports an average SMT-speedup of 28%, falling between the speedups reported by Fixed (13%) and Variable (35%). FIESTA also significantly reduces the load imbalance observed in the Fixed experiment. Fixed yields an average load imbalance of 55%, with a maximum of 96% in the *mcf*/*mesa* (*Mm*) pairing. FIESTA yields a maximum imbalance of 65% and an average of only 21%. Comparing the two tells us that 62% of the imbalance in the Fixed experiment is sample imbalance, the remainder is schedule imbalance.

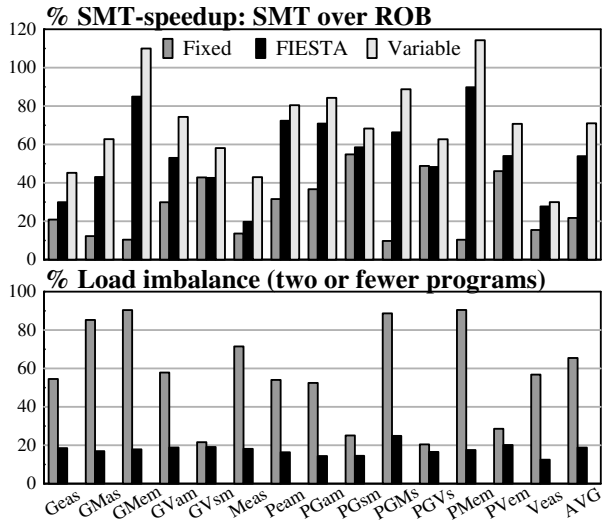


Figure 4: Four-thread experiments.

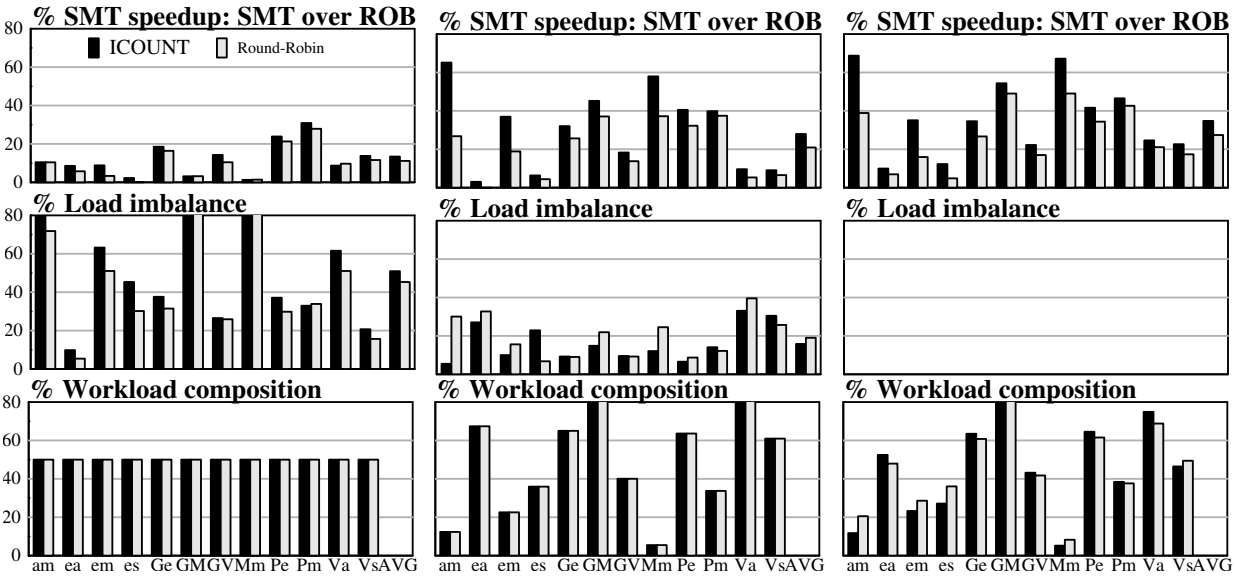
No contention or symmetric contention. Results for individual benchmark pairings can be understood by looking at SMT-speedup, imbalance, and (especially) workload composition for the three methodologies. We begin with the workload composition graph. FIESTA workload compositions correspond to idealized contention-free concurrent execution. Programs are represented in the workload according to their standalone performance—programs with low standalone throughputs like *mcf* (*M*) comprise smaller percentages (by instructions) of their corresponding workloads. For a given pairing, Variable having a similar composition to FIESTA indicates either little or symmetric contention. In these pairings, FIESTA shows low (schedule) imbalance, and FIESTA and Variable SMT-speedups agree closely. FIESTA SMT-speedups agree with those of Fixed if the paired benchmarks have roughly equal standalone performance. In pairs with drastically different standalone throughputs (*e.g.*, *mcf*/*mesa* (*Mm*)), Fixed results in a significant amount of sample imbalance, which in turn drives SMT-speedup towards zero.

Asymmetric contention. A discrepancy between FIESTA and Variable workload compositions is an indication of asymmetric contention. In most of these cases, Variable biases the workload in favor of the slower program, pushing the composition closer to a 50/50 split. We traced this effect to the use of capped partitioning for the stream buffers. Executing alone, a program that benefits from prefetching can use all eight buffers. However, when paired with another program, it is limited to using six buffers, even if the second program is not using the other two. *art* is one program that benefits from being able to use all eight stream buffers and pairing it with another program reduces its performance. In Variable, *art*'s reduced performance reduces its representation in the workload. Reducing the representation of a program whose performance degrades more in favor of a program whose performance degrades less inflates SMT-speedup. In pairings with asymmetric contention (*e.g.*, most *art* pairings) FIESTA agrees more closely with Fixed than with Variable. The *art* pairings illustrate the subtle pitfall of variable-workload methodologies. While metrics like SMT-speedups prevent variable-workload schemes from silently favoring programs with higher standalone performance, they do not prevent them from favoring threads that "benefit" (*i.e.*, suffer less) from asymmetric contention.

Opposing sample and schedule imbalance. Two interesting pairings are *equake*/*art* (*ea*) and *vortex*/*swim* (*Vs*). In these, FIESTA reports lower SMT-speedups and higher load imbalance than Fixed. These pairings are instances in which sample and schedule imbalance act in opposite directions. In Fixed, sample imbalance "favors" the slower application. And as we saw previously, capped resource partitioning usually favors the slower application as well by capping the resource consumption and performance of the faster application. In these cases, however, schedule imbalance favors the faster application. For instance, both *art* and *equake* use stream buffers heavily, with *art* (the slower application) suffering a greater loss from reduced stream buffer allocations. By eliminating sample imbalance, FIESTA exposes the schedule imbalance that favors *equake*.

A word about fairness. Our interpretation of the results so far has not used the fairness metric as an explanatory tool. Fairness is not a stable or even particularly meaningful metric in the presence of load imbalance. However, in FIESTA, all load imbalance is schedule imbalance. This makes FIESTA schedule imbalance a good fairness metric on its own. Experimentally, the correlation coefficient between load balance in FIESTA and traditional fairness for FIESTA is 0.97.

Four-thread workloads. Figure 4 shows results for four-thread workloads. The load imbalance graph plots the percentage of time during which two or fewer programs are executing. On average, the trends match those seen in the two-thread workloads. The Fixed experiments report 21% speedup and 64% load imbalance. The Variable experiments report a 71% speedup. The FIESTA experiments report a speedup of 54%—about two-thirds of the way from Fixed to Variable—with a proportional decrease in imbalance, down to 19%.



(a) Fixed (b) FIESTA (c) Variable

Figure 5: Same-architecture study: comparing thread-scheduling policies ICOUNT and Round-Robin.

3.2. Same-Architecture Studies

FIESTA should be ideal for "policy" studies in which different multi-program experiments share the same underlying single-program baseline. A FIESTA workload generated on this common baseline is sample-balanced by construction in all experiments. A common study of this kind compares thread scheduling policies [19]. In Figure 5, we use the three workload methodologies to compare ICOUNT and Round-robin.

All three methodologies agree that ICOUNT yields higher SMT-speedup than round-robin, although Fixed reports a 3% throughput advantage, whereas FIESTA and Variable measure ICOUNT's advantage at 7%. Overall, Fixed depresses total throughput and SMT-speedup while Variable inflates them.

As expected, FIESTA reduces load imbalance as a whole. Interestingly, the Fixed experiments show higher imbalance for ICOUNT than for Round-robin. Round-robin effectively interleaves threads instruction-by-instruction, "matching" a Fixed workload in which all applications execute the same number of instructions. Meanwhile, ICOUNT attempts to interleave applications cycle-by-cycle—actually, all thread scheduling policies *attempt* to do this—matching a workload in which applications execute for the same number of cycles, as in FIESTA.

The workload composition graphs show the potential hazard of using Variable workloads for direct comparisons. Several program pairs—*art*/*mesa* (*am*), *equake*/*swim* (*es*), and *vortex*/*art* (*Va*)—produce workloads whose composition varies by more than 10% from one experiment to another.

3.3. Cross-Architecture Studies

Cross-architecture studies—in which different experiments have different single-program baselines—present a challenge for FIESTA. Because standalone program performance may be different on the different baselines, a FIESTA workload created on one architecture may not be sample-balanced when used in experiments with different architectures.

We argue that FIESTA provides reasonable workloads for these studies as well. It does not matter which architecture the FIESTA workloads are constructed with as long as the same workloads are used consistently. Here is the intuition. In workload jargon, running a FIESTA workload on an architecture other than the one on which it was created has the effect of increasing sample imbalance. However, this effect is much smaller than FIESTA's sample-imbalance reduction effect. A different baseline architecture may affect the IPC of a given program by a factor of 2 or 3. However, the "natural" IPC of different programs may differ by a factor of 15 or more, as it does for *mcf* (0.2) and *mesa* (3.3). Furthermore, a different architecture likely affects the performance of all programs in the same direction. If it increases the IPC of both programs by a factor of 2, then sample imbalance between them remains 0%.

FIESTA workload robustness. Figure 6 shows

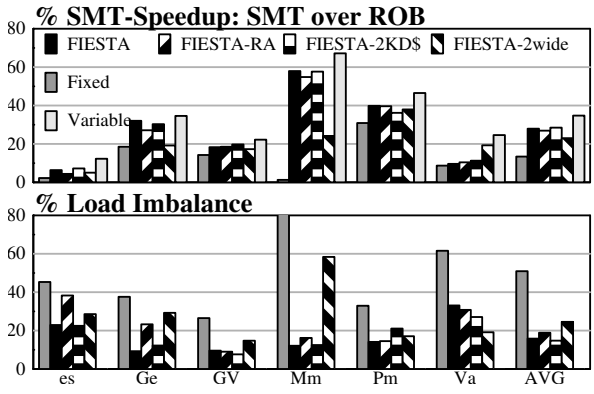


Figure 6: FIESTA workload robustness to architecture changes. All experiments measure "vanilla" SMT relative to vanilla ROB. We use Fixed, Variable, and four FIESTA workloads. The FIESTA workload is generated on vanilla ROB and is sample-balanced by construction. The other FIESTA workloads are generated on a 2-way issue processor, a processor with a 2KByte data cache, and a Runahead processor.

SMT-speedup and load imbalance for several experiments using our "central" SMT configuration. In addition to Fixed, Variable, and (same-architecture) FIESTA workloads, we also use three additional FIESTA workloads created on other baselines: FIESTA-2wide is created on a 2-way superscalar processor, FIESTA-2KD$ on a processor with a 2KByte data cache, and FIESTA-RA on a Runahead processor. Our central configuration is 4-way superscalar with a 64-KByte data cache and no Runahead execution. Runahead improves performance by 5% on average, but this improvement is concentrated in three programs: *equake* (39%), *art* (4%), and *mcf* (2%). A 2-KByte data cache reduces IPC by 12% with a range of 0% (*equake*) to 26% (*perl*). 2-way superscalar issue reduces IPC by 30% with a range of 3% (*art*) to 44% (*mesa* and *vortex*).

On average, experiments using (same-architecture) FIESTA workloads report SMT-speedups that are 2%, 1%, and 6% higher than those using FIESTA-RA, FIESTA-2KD$, and FIESTA-2wide workloads. FIESTA-2wide has the highest difference because this configuration has the biggest relative impact on standalone execution times (30% on average). That SMT-speedup differs by *only* 6% reflects the fact that 2-way superscalar execution changes the standalone performance of all programs in the same direction and therefore "re-introduces" a sample imbalance of only 9% (this is calculated as the difference between FIESTA imbalance which is "pure" schedule imbalance and FIESTA-2wide imbalance). The more significant point is that despite the discrepancy is standalone performance, the FIESTA-2wide workload yields more balanced runs and more accurate SMT-speedups than a Fixed workload.

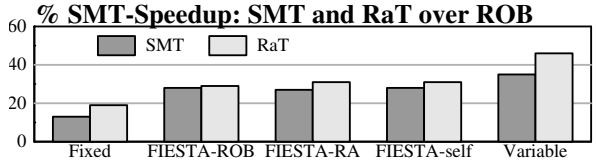


Figure 7: Cross-architecture study: average SMT-speedup over ROB using different methodologies.

SMT vs. RaT. Figure 7 compares multi-program architectures with different single-program baselines: SMT's baseline is a "vanilla" ROB processor, the single-program baseline for RaT (Runahead Threads) [13] is a Runahead processor. We compare *average* SMT-speedups over vanilla ROB using different workload methodologies. For the FIESTA-self methodology, SMT experiments using FIESTA-ROB workloads while RaT experiments use FIESTA-RA workloads.

Following the trends we have seen several times, Fixed deflates SMT-speedups while Variable tends to inflate them. However, in this particular case, both Fixed and Variable tend to inflate RaT's advantage over SMT. Recall, the standalone advantage of Runahead over ROB is 5%. We would expect the advantage of RaT over SMT to be somewhat less than this because there is some obvious overlap between Runahead execution and multi-threading and little obvious synergy. However, Fixed reports RaT's advantage as 6% and Variable as 11%. Interestingly, they do this in different ways. Variable inflates RaT's speedups using the standard over-sampling pathology. Fixed deflates SMT's speedups by virtue of high load imbalance, much which is due to programs that benefit from Runahead execution. By accelerating these programs, RaT removes the Fixed imbalance and exposes the SMT-speedup that is provided by multi-threading. The FIESTE workloads not only show more moderate performance, they also report RaT's advantage as being in the range of 1–4%. This result is more consistent with what we know about Runahead execution performance. Overall, FIESTA-ROB and FIESTA-RA workloads generate speedups that differ by only 1%.

These experiments suggest (although do not *prove*) that FIESTA workloads can be used as a basis for directly comparing different architectures. FIESTA may not be an ideal methodology for such studies, but it is better than existing fixed-workload (which suffer much higher sample imbalance) schemes. If drastic architectural differences make sample imbalance a concern with FIESTA, the effect can be measured. The experiments

can be run from workloads generated on both architectures, and the similarity or difference in the results can provide confidence in their significance.

4. Other Multi-Program Workload Issues

FIESTA addresses issues of sample-balance in multi-program workload construction. There are several other related and orthogonal issues as well.

Representative samples of individual programs. Real world usage scenarios use complete programs, not program samples. Any sampling scheme should strive to use samples that faithfully represent the complete program. It is straightforward to obtain representative *instruction-based* samples, either using uniform distribution [24] or phase analysis [11]. Some variable-workload methodologies [17, 23] attempt to preserve individual program representativeness by restarting samples of fast programs when slower programs finish their sample.

FIESTA relies on time-based samples which—when applied with a uniform instruction distribution—oversample fast program regions relative to slow ones. One way to marry FIESTA with representative program samples is to modify SimPoint to produce a representative sample that executes for a specified number of cycles rather than for a specified number of instructions. We leave this for future work.

Representative or interesting program combinations. An issue that is largely orthogonal to FIESTA is choosing representative and interesting combinations of programs and program phases that should be executed concurrently [21].

Multi-program workloads that include multi-threaded programs. One important extension to FIESTA would allow workloads to include multi-threaded programs. Inherently, FIESTA should be applicable to multi-threaded programs, given that it is possible to create time-based samples for them. FIESTA would ignore any internal imbalance that exists within the program and consider only imbalance between programs as a whole.

Experiments on real hardware. We presented and evaluated FIESTA as a workload methodology for simulation experiments. However, assuming proper support for executing and measuring pre-defined program samples, it is applicable to experiments on real hardware as well [17].

Acknowledgements

We thank the reviewers for their comments. This work was supported by NSF award CCF-0541292 and by a grant from the Intel Research Council.

References

[1] F. Cazorla, E. Fernandez, A. Ramirez, and M. Valero. Optimizing Long-Latency-Load Aware Fetch Policies for SMT Processors. *IJHPCN-2004*, Apr. 2004.

[2] F. Cazorla, A. Ramirez, M. Valero, and A. Fernandez. Dynamically Controlled Resource Allocation in SMT Processors. In *Proc. MICRO-37*, Dec. 2004.

[3] Z. Chishti and T. Vijaykumar. Wire Delay is Not a Problem for SMT (in the Near Future). In *Proc. ISCA-31*, Jun. 2004.

[4] A. El-Moursi and D. Albonesi. Front-End Policies for Improved Issue Efficiency in SMT Processors. In *Proc. HPCA-9*, Feb. 2003.

[5] S. Eyerman and L. Eeckhout. A Memory-Level Parallelism Aware Fetch Policy for SMT Processors. In *Proc. HPCA-13*, Feb. 2007.

[6] R. Gabor, S. Weiss, and A. Mendelson. Fairness and Thruput in Switch on Event Multithreading. In *Proc. MICRO-39*, Dec. 2006.

[7] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: the Potential for Processor Power Reduction. In *Proc. MICRO-36*, Dec. 2003.

[8] R. Kumar, N. Jouppi, and D. Tullsen. Conjoined-Core Chip Multiprocessing. In *Proc. MICRO-37*, Dec. 2004.

[9] K. Luo, M. Franklin, S. Mukherjee, and A. Seznec. Boosting SMT Performance by Speculation Control. In *Proc. IPDPS-15*, Apr. 2001.

[10] K. Luo, J. Gummaraju, and M. Franklin. Balancing Throughput and Fairness in SMT Processors. In *Proc. ISPASS-2008*, Nov. 2001.

[11] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder. Using SimPoint for Accurate and Efficient Simulation. In *Proc. SIGMETRICS-2003*, Jun. 2003.

[12] S. Raasch and S. Reinhardt. The Impact of Resource Partitioning on SMT Processors. In *Proc. PACT-12*, Sep. 2003.

[13] T. Ramirez, A. Pajuelo, O. Santana, and M. Valero. Runahead Threads to Improve SMT Performance. In *Proc. HPCA-13*, Feb. 2008.

[14] Y. Sazeides and T. Juan. How To Compare The Performance of Two SMT Microarchitectures. In *Proc. ISPASS-2001*, Nov. 2001.

[15] A. Snavely and D. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. In *Proc. ASPLOS-9*, Oct. 2000.

[16] S. Subramaniam, M. Prvulovic, and G. Loh. PEEP: Exploiting Predictability of Memory Dependences in SMT Processors. In *Proc. HPCA-13*, Feb. 2008.

[17] N. Tuck and D. Tullsen. Initial Observations of the Simultaneous Multithreading Pentium 4 Processor. In *Proc. PACT-12*, Sep. 2003.

[18] D. Tullsen and J. Brown. Handling Long-Latency Loads in a Simultaneous Multithreading Processor. In *Proc. MICRO-34*, Dec. 2001.

[19] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proc. ISCA-23*, May 1996.

[20] E. Tune, R. Kumar, D. Tullsen, and B. Calder. Balanced Multithreading: Increasing Throughput via a Low Cost Multithreading Hierarchy. In *Proc. MICRO-37*, Dec. 2004.

[21] M. van Biesbrouck, L. Eeckhout, and B. Calder. Representative Multi-Program Workloads for Multithreaded Processor Simulation. In *Proc. IISWC-2007*, Sep. 2007.

[22] K. van Craeynest, S. Eyerman, and L. Eeckhout. MLP-Aware Runahead Threads in a Simultanenous Multithreading Processor. In *Proc. HiPEAC-4*, Jan. 2009.

[23] J. Vera, F. Cazorla, A. Pajuelo, O. Santana, E. Fernandez, and M. Valero. FAME: FAirly MEasuring Multithreaded Architectures. In *Proc. PACT-16*, Sep. 2007.

[24] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. SMARTS: Accelarating Microarchitectural Simulation via Rigorous Statistical Sampling. In *Proc. ISCA-30*, Jun. 2003.