Multi-Program Benchmark Definition

Adam N. Jacobvitz, Andrew D. Hilton, Daniel J. Sorin
Department of Electrical and Computer Engineering
Duke University, Durham, North Carolina, USA, 27708
adam.jacobvitz@sandisk.com, adhilton@ee.duke.edu, sorin@ee.duke.edu

***Abstract*—Although definition of single-program benchmarks is relatively straight-forward—a benchmark is a program plus a specific input—definition of multi-program benchmarks is more complex. Each program may have a different runtime and they may have different interactions depending on how they align with each other. While prior work has focused on sampling multi-program benchmarks, little attention has been paid to defining the benchmarks in their entirety.**

In this work, we propose a four-tuple that formally defines multi-program benchmarks in a well-defined way. We then examine how four different classes of benchmarks created by varying the elements of this tuple align with real-world use-cases. We evaluate the impact of these variations on real hardware, and see drastic variations in results between different benchmarks constructed from the same programs. Notable differences include significant speedups versus slowdowns (*e.g.*, +57% vs -5% or +26% vs -18%), and large differences in magnitude even when the results are in the same direction (*e.g.*, 67% versus 11%).

I. Introduction

With multi-core processors now ubiquitous and cloud computing becoming more prevalent, architects are increasingly evaluating their micro-architectural ideas using multi-program workloads. To make fair and insightful comparisons between systems, architects need appropriate multi-program benchmarks for their experiments. That is, architects need a set of individual programs and a methodology for running these programs during experiments; taken together, these constitute the *benchmark definition*.

Although benchmark definition is trivial for single-program workloads—just run the single program to completion—defining a multi-program benchmark is more complicated. The reason for this complexity is that there are many ways in which one can run multiple programs. For example, we could run each program once on its own core and run until the first (or last) program finishes. Instead, we could run each program on its own core as many times as possible until a pre-specified amount of time has elapsed. Another possibility is to run a set of programs and launch them in FIFO order to the next available core. There are numerous other possibilities, many of which are reasonable representations of real-world systems.

Benchmark definition is crucial for multi-program experiments because it can profoundly impact the results. Even with the same program pairings, different multi-program benchmark definitions can have qualitatively different experimental results, leading to different conclusions. Section IV shows how different benchmark definitions with the same program pairings affect whether we see a speedup or a slowdown between two systems, or radically different speedups.

The experimental differences between benchmark definitions arise due to the interactions between concurrently running programs. One issue that arises in multi-program benchmarks is *load imbalance*, in which different programs run for different amounts of time. Assume program A runs for 1 second and program B runs for 10 seconds. If the benchmark definition is to run each program once until they both complete, then 90% of the experiment will measure program B running alone. Benchmark designers must take care to ensure that the load imbalance in their benchmarks accurately models the load imbalance in the systems their work targets.

Interactions between programs running concurrently in a multi-program benchmark include not just load imbalance but also contention for shared resources. The resource usages of the programs—and thus their contention patterns—vary across the execution of a program, primarily as it transitions from one phase of execution to another [19]. This phase-dependent contention behavior means that the *program alignment*—the relative position of the two programs in their dynamic instruction streams—affects the performance characteristics of the multi-program benchmark. Consequently, a benchmark definition must account for the variety of alignments that occur in the scenarios that the benchmark intends to model.

Given that benchmark definition is crucial for multi-program experiments, it is perhaps surprising that there is no universally accepted way of precisely (*i.e.* formally) defining or even *describing* multi-program benchmarks. Research papers textually describe their multi-program benchmarks—although often without sufficient detail for the reader to reproduce the results. Table I presents a sample of multi-program benchmark descriptions from a variety of research papers.

Examining these descriptions, it is apparent that benchmark definition methodologies differ drastically between papers. One issue that varies considerably is the benchmark duration. Some papers define the benchmark duration by the number of instructions executed [4], [6], [9], [12], [16], though some are not clear whether that is instructions for each program, total instructions between all programs, when one program reaches a particular number of instructions, or some other condition. Other papers define the benchmark duration by the number of cycles executed [15], [17] or until the IPC converges [10].

Paper	Multiprogram Benchmark Description
"Balancing DRAM Locality and Parallelism in Shared Memory CMP Systems" [6]	"..., we use SimPoint and determine a representative 200 million instruction region from each application."
"CPROB: Checkpoint Processing with Opportunistic Minimal Recovery" [4]	"Our multi-program workload metholodgy is FIESTA. From each program we choose 50 samples, each of which runs for 5 million cycles when executed standalone. A multi-program run executes the samples from the different programs pair-wise."
"Effective Management of DRAM Bandwidth in Multicore Processors" [12]	"...the workloads are simulated in detail for 100 million instructions."
"Fair Queuing Memory Systems" [9]	"We use twenty 100 million instruction SPEC benchmark sampled traces that have been verified to be statistically representative of the entire SPEC application."
"A Flexible Heterogeneous Multi-core Architecture" [10]	"we use the methodology proposed in [FAME: FAirly MEasuring Multithreaded Architectures]...we used a Maximum Allowable IPC Variance (MAIV) of 5%."
"MISE: Providing performance predictability and improving fairness in shared main memory systems" [17]	"We extract a representative phase of each benchmark using PinPoints and run that phase for 200 million cycles."
"Symbiotic Jobscheduling for a Simultaneous Multithreading Processor" [15]	"Every 5 million cycles,..., the jobscheduler receives a clock pulse; if runnable jobs are available that were not scheduled during the previous timeslice, it swaps out one or more of the jobs that ran in the last timeslice, replacing these with jobs that did not."
"Symbiotic Jobscheduling with priorities for a Simultaneous Multithreading Processor" [16]	"The benchmarks were fast-forwarded to get out of the startup phase before being simulated for 250 million instructions times the number of threads being simulated."

Table I: Selection of multiprogram benchmark descriptions used in recent work.

Another issue that varies across multi-program benchmark definitions is the mapping of programs to cores. A benchmark could assign all programs of a given type to a given set of cores, or it could assign programs in a FIFO manner to the next available core.

To enable better experimental evaluations and higher reproducability of others' work, we argue for a framework for precisely defining multi-program benchmarks. With such a framework, experimenters could clearly and concisely describe their benchmarks, enabling peer-reviewers to assess the appropriateness of the benchmarks and other researchers to re-execute the benchmark exactly. However, we are *not* arguing that there should be a single benchmark definition for all experiments, because the benchmark must be appropriate for the experiment. A multi-program benchmark appropriate to a web-server would be inappropriate for a real-time control system, even if the individual programs comprising it were appropriate to both. None of the papers shown in Table I (or any others that we know of) explicitly discuss how their benchmark definitions align with the behavior of the systems they target.

In this work, we develop a framework for precisely and unambiguously defining multi-program benchmarks. A benchmark defined using our framework leaves no question as to how the programs were co-executed, making it directly reproducible. At the same time, our framework is flexible enough to allow the wide variety of benchmark definitions required to match the wide array of possible system usage models. We show how to use our framework to define benchmarks for some system usage models (*e.g.*, highly-utilized servers, servers with constrained scheduling, etc.). Although we obviously cannot show all possible benchmark definitions that can be specified in our framework, the examples provide a clear demonstration of its usefulness and importance. Architects targetting other systems can use our framework to define whatever benchmarks are appropriate, and can succinctly describe them with enough precision that others can fully understand them.

The primary contributions of this work are:

- A framework for precisely defining and unambiguously describing multi-program benchmarks.
- Examples of how the framework can be applied to define benchmarks appropriate to experimentation for some real system usage models.
- Experimental demonstration of the impact that benchmark definition has on experimental results.

II. The Elements of a Multi-Program Benchmark Experiment

Before we present our new framework for precisely defining multi-program benchmarks, it is important and insightful to clearly separate the elements of an experiment in order to understand the critical role played by benchmark definition. We now present the steps necessary to construct a rigorous scientific experiment with multi-program benchmarks.

Step 1: Benchmark program selection. A multi-program benchmark involves running multiple single-program benchmarks (program + input), and we distinguish between the selection of these single-program benchmarks (which is Step #1) and how they are co-executed (Step #2). For Step #1, an experimenter simply chooses how many programs to run, which programs to run, and the appropriate input data-sets for each.

Step 2: Multi-program benchmark definition. This step is the focus of this paper and where our contribution lies. We develop a framework for precisely specifying how to run the programs that were selected in Step #1.

Step 3: Evaluation metric selection. Experimenters choose a metric that is appropriate to the issue they are exploring. There are metrics for performance, power, reliability, etc. Metric selection is a very important issue that is the subject of much debate [2], [3], [8], [13], but it is mostly orthogonal to the focus of this paper. That is, a well-defined benchmark could be used to evaluate whatever metric is chosen in Step #3.

However, the choice of metric could become somewhat entangled with benchmark definition (but orthogonal to the

general formalism for benchmark definition that we present). As we will see shortly, different multi-program benchmarks may hold different aspects (*e.g.*, programs executed, runtime, energy consumed, etc.) constant. In a benchmark that holds "programs executed" constant, simply measuring "runtime" gives a meaningful performance metric; lower runtime equals higher performance. However, in a benchmark that holds "runtime" constant, measuring runtime is of course, silly. Instead, higher performance comes from a higher number of jobs executed in the fixed time. Of course, if the ratio of the programs co-executed changes, then some system model dependent notion of total work must be applied to combine them.

Step 4 (optional): Benchmark sampling. To reduce the time required for the experiment, the experimenter may choose to sample the benchmark's execution. Sampling methodology is an active area of research (*e.g.*, [5], [11], [13], [18], [19], [22]), but it is largely orthogonal to the focus of this paper. To sample a benchmark requires first defining the benchmark to be sampled. (Put another way, to quantify sampling error, one must compare the results of the sampled benchmark to a baseline unsampled benchmark.) Some researchers have developed sampling methodologies whereby they **start** with the sampled benchmark and then they may or may not infer the definition of the unsampled benchmark. We note that sampling is not entirely orthogonal to benchmark definition in that some sampling methodologies may not be applicable to all benchmark definitions.

III. Benchmark Definition

A benchmark's definition must specify all of the parameters required to ensure that the benchmark can be used to make meaningful comparisons. Typically, these comparisons are in terms of one or more metrics (*e.g.*, time, or energy) on two or more architectures. Executing a well-defined benchmark on each architectures allows for direct comparison of the metrics measured during the benchmark's execution. By contrast, ill-defined benchmarks may yield invalid comparisons.

For single-program experiments, benchmark definition is well understood—a benchmark is the execution of a particular program on a specific input. Running the same program on the same input produces results that are directly comparable across architectures. If `lbm.ref` takes 30 seconds on architecture α and 60 seconds on architecture β, we can typically conclude that architecture α is faster than architecture β on `lbm.ref`. Of course, in doing this comparison, one must be cautious of the broader experimental methodology—*e.g.*, performing multiple runs, determining significance of results relative to experimental error, eliminating other interference—however, these concerns are orthogonal to benchmark definition.

The challenge we address in this paper is the precise and unambiguous definition of multi-program benchmarks. We have developed a framework in which we define a multi-program benchmark as a 4-tuple: $\langle \mathcal{B}, \mathcal{T}, \mathcal{F}, \mathcal{S}_0 \rangle$. The first element, $\mathcal{B}$, is the set of single-program benchmarks (programs plus specific inputs) that are combined to make the multi-program benchmark. For example, $\mathcal{B}$ might be $\{$`lbm.train`, `gcc.ref`$\}$. In this paper, we consider only single-threaded programs; defining multi-program benchmarks consisting of multi-threaded programs is future work.

The second element of a multi-program benchmark definition is $\mathcal{T}$, the terminating condition for the benchmark, which specifies when the benchmark is complete. For single-program benchmarks, the terminating condition for the benchmark is implicit and simple: when the program of interest exits, the benchmark ends. For multi-program benchmarks, however, the terminating condition may take a variety of forms, such as "200 `lbm`s and 100 `gcc`s have completed" or "the benchmark has executed for 20 minutes."

The third element of the multi-program benchmark definition is $\mathcal{F}$, the *selection function*, which decides which member of $\mathcal{B}$ to run on a particular core when that core becomes idle. This function may require some state (*e.g.*, to behave as if selecting jobs from a queue), and we denote this state as $\mathcal{S}$. $\mathcal{F}$ is a function of the current state and the core (specified as a natural number) for which it is selecting. Given those inputs, $\mathcal{F}$ then produces a new state (to be used in the next invocation of $\mathcal{F}$), as well as the single-program benchmark (if any) to run on that core. The function may return $\perp$, selecting no member of $\mathcal{B}$, and instead leaving that core idle. Formally, $\mathcal{F}$ has type $\mathcal{S} \times \mathbb{N} \Rightarrow \mathcal{S} \times \mathcal{B}_\perp$. From a mathematical perspective, whenever one or more cores are idle (including at the start of the benchmark), $\mathcal{F}$ is evaluated on each idle core in order from lowest to highest number. (Of course, actually running a benchmark would involve a more efficient implementation.)

The fourth element of the multi-program benchmark definition, $\mathcal{S}_0$ is the initial state for $\mathcal{F}$. The type of $\mathcal{S}_0$ is clearly dependent on the way in which we specify the state used by $\mathcal{F}$. In this paper, we focus on state that is represented using one or more queues, thus the initial state $\mathcal{S}_0$ is the initial state of those queues.

A. Space of Multi-Program Benchmarks

Our framework for defining multi-program benchmarks provides the flexibility to precisely describe a wide range of benchmarks. In fact, the space of possible benchmarks is unbounded, even if we fix $\mathcal{B}$. However, not all benchmarks are interesting or representative of real-world use cases. For example, selecting $\mathcal{T}$ = "two programs execute a system call within 10 cycles" is well-defined, but does not make intuitive sense. Likewise, as $\mathcal{F}$ is an arbitrary function, it could be illogical relative to typical ways to execute programs. While our framework *can* be used to define such benchmarks, they are uninteresting to examine.

We focus our explorations on options for three of the four elements of the benchmark definition tuple: $\mathcal{T}$, $\mathcal{F}$, and $\mathcal{S}_0$. The choice of set $\mathcal{B}$ is orthogonal to our work. For $\mathcal{T}$, we explore both work-based and time-based criteria for termination. For

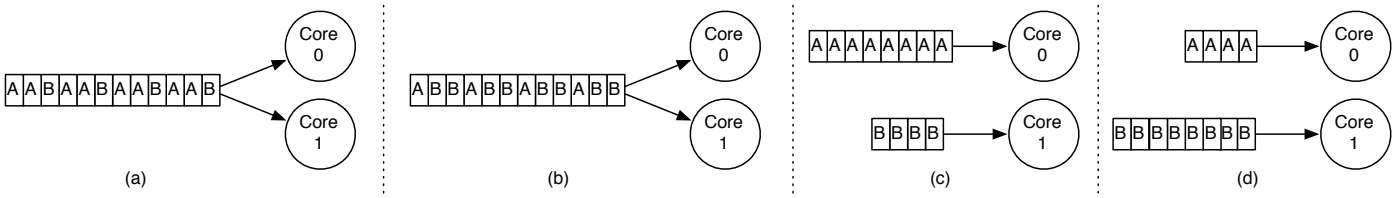

Figure 1: Four benchmarks which all have $\mathcal{B} = \{\mathrm{A}, \mathrm{B}\}$, $\mathcal{T}$ =finish all jobs. Benchmarks (a) and (b) have $\mathcal{F}(q, n) = \langle \mathrm{tail}(q), \mathrm{head}(q) \rangle$, while benchmarks (c) and (d) have $\mathcal{F}(q, n) = \langle \mathrm{tail}(q[n]), \mathrm{head}(q[n]) \rangle$. Benchmark (a) has $\mathcal{S}_0 = \mathrm{AABAABAABAAB}$, (b) has $\mathcal{S}_0 = ABBABBABBABBABB$, (c) has $\mathcal{S}_0 = [\mathrm{AAAAAAAA}, \mathrm{BBBB}]$ and (d) has $\mathcal{S}_0 = [\mathrm{AAAA}, \mathrm{BBBBBBBB}]$.

$\mathcal{F}$, we explore functions that use one or more queues as their state. Thus $\mathcal{S}_0$ is the initial state of these queues.

Given our focus on queue-based $\mathcal{S}$, the choices for $\mathcal{T}$ and $\mathcal{F}$ suggest we should examine four classes from four "quadrants"—formed by the cross-product of $\mathcal{T}$={work-based, time-based} and $\mathcal{F}$={one queue per core, one queue shared by all cores}. We now examine each of these classes and, for each one, describe what real-world use case it represents.

B. Work-Based / Single Queue

The first class we explore consists of benchmarks that run a specified amount of work (e.g., 7 executions of program A and 3 executions of program B) and use a single queue of jobs that is shared among all cores. The initial state of the queue is the set of all jobs (single-threaded programs) that must be completed for the benchmark to be complete. The program selection function dispatches jobs to cores in a FIFO fashion. We refer to this class of benchmarks as *JO* ("Complete all *J*obs, *O*ne Queue").

This benchmark class corresponds to highly-utilized servers. We can choose the initial state of the queue to contain the ratio of job types we expect to arrive at our server. For example, we can initialize the queue to hold jobs of type A, type B, and type C in a ratio of 2:1:3 or 2:5:1 or whatever is appropriate for the system being studied. Importantly, in this class, the rate at which the system completes jobs of a particular type does not affect the ratio of work it must complete.

Benchmarks in this class may be precisely defined in our benchmark definition framework by letting the state maintained for $\mathcal{F}$ be a FIFO queue whose elements are members of $\mathcal{B}$. Specifically, $\mathcal{F}(q, n) = \langle \mathtt{tail}(q), \mathtt{head}(q) \rangle$, where $\mathtt{tail}(q)$ returns a queue just like q, except the first element is removed, and $\mathtt{head}(q)$ returns the first element of q[1]. For this class of benchmarks, $\mathcal{T}$ = "all jobs from the initial queue have been executed." Varying $\mathcal{B}$ and $\mathcal{S}_0$ produce different benchmarks within this class.

Experimenters may devise the $\mathcal{S}_0$s for their benchmarks in a variety of ways (randomly, from real system traces, etc.). In this work, we do not propose a specific technique for constructing $\mathcal{S}_0$ (especially as there is no one *right* answer). Instead, the important consideration from our perspective is that $\mathcal{S}_0$ must be precisely defined, and remain fixed across architectures being compared. Changing $\mathcal{S}_0$ results in a different benchmark. Sections IV-C and IV-D evaluate the impact of the program ratio and ordering in $\mathcal{S}_0$ respectively.

Figure 1 illustrates four different benchmarks, all of which have $\mathcal{B} = \{\mathrm{A}, \mathrm{B}\}$ (where A and B are single program benchmarks), and $\mathcal{T}$ =finish all jobs. Of these, the left two benchmarks, (a) and (b), fall into this *JO* class of benchmarks. These two benchmarks differ in their values of $\mathcal{S}_0$. The two different values of $\mathcal{S}_0$ give the two benchmarks different ratios of jobs of type A to jobs of type B, possibly resulting in drastically different performance characteristics. The distinction between the benchmarks shown in Figure 1 (a) and (b) illustrates a point of caution in designing multi-program benchmarks—even though they may appear quite similar, they are two distinct benchmarks.

C. Work-Based / Per-Core Queues

In this class of benchmarks, we still have a fixed amount of work to perform but now have an array of per-core queues, indexed by core number, instead of a single queue. Specifically, we fix $\mathcal{F}(q, n) = \langle \mathrm{tail}(q[n]), \mathrm{head}(q[n]) \rangle$, and $\mathcal{T}$ = "all jobs from the initial queues have been executed." We refer to this class of benchmarks as *JM* ("Complete all *J*obs, *M*ultiple Queues").

This class of benchmarks represents server workloads in which there are constraints on how jobs may be scheduled onto the cores. One such constraint may arise from heterogeneity in the cores' ISAs (*e.g.*, Cell [7]); only jobs whose ISA match a particular core may be scheduled to it. Another scheduling constraint may arise due to quality-of-service (QoS) requirements (*e.g.* Paragon [1]), possibly where one type of jobs belongs to a datacenter customer paying a premium for dedicated use of some set of cores.

Figure 1 (c) and (d) depict two benchmarks in this class that have different $\mathcal{S}_0$. In both of these benchmarks, Core 0 runs only As, and Core 1 runs only Bs. These two benchmarks will primarily differ in their *load imbalance*—when one core has finished all of its jobs, but the other has not.

Even though benchmarks (a) and (c) have the same A-to-B ratio (as do (b) and (d)), they will exhibit different characteristics. Notably, (c) and (d) will never run A one on core with A on the other core also nor B with B (both of which

[1]Note that $\mathtt{head}(\emptyset) = \perp$ and $\mathtt{tail}(\emptyset) = \emptyset$.

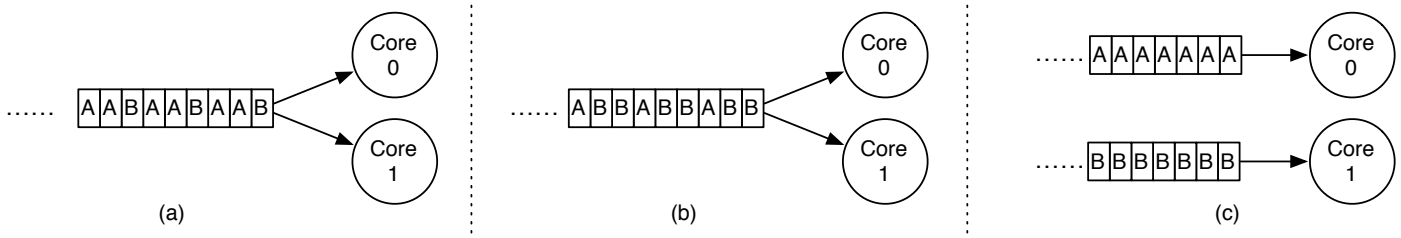

Figure 2: Three benchmarks which all have $\mathcal{B} = \{\mathrm{A}, \mathrm{B}\}$, $\mathcal{T}$ =run for X minutes. Benchmarks (a) and (b) have $\mathcal{F}(q, n) = \langle \mathrm{tail}(q), \mathrm{head}(q) \rangle$, while benchmark (c) has $\mathcal{F}(q, n) = \langle \mathrm{tail}(q[n]), \mathrm{head}(q[n]) \rangle$. Here, the queues are infinite, and benchmark (a) has $\mathcal{S}_0 = \ldots \mathrm{AABAAB}$, (b) has $\mathcal{S}_0 = ...ABBABB$, and (c) has $\mathcal{S}_0 = [\ldots \mathrm{AAAA}, \ldots \mathrm{BBBB}]$.

can occur in (a) and (b)). Such pairings may exhibit different performance characteristics than A/B pairings. For example, suppose the two cores share a cache and A has a large cache footprint, but B has a small one. A/B pairings may fit entirely in the cache, whereas A/A pairings may exhibit a significant number of cache misses. The (a) and (b) benchmarks will also exhibit different load imbalance from their (c)/(d) counterparts.

D. Time-Based / Per-Core Queues

A third class of benchmarks arises when particular job types are pinned to particular cores ($\mathcal{F}(q, n) = \langle \mathrm{tail}(q[n]), \mathrm{head}(q[n]) \rangle$, as in *JM*), for a fixed amount of time ($\mathcal{T}$ = "run for X minutes"). We refer to this class as *TM* ("Run for a fixed *T*ime, *M*ultiple Queues"). Unlike the previous classes, the queues for this class of benchmarks have *infinite* length, so that the jobs never run out; instead, the benchmark ends when a certain amount of time expires. Figure 2 part (c) illustrates benchmarks in this class. Unlike the previous class of benchmarks, there is no notion of altering the ratio of the benchmarks by changing $\mathcal{S}_0$. Instead, the ratio of the benchmarks is defined by the relative rate at which the cores execute them. This distinction is important because it means that the A-to-B ratio is *architecturally determined*—the ratio on architecture α may be different than the ratio for the *same* benchmark on architecture β.

This class of benchmarks aligns with systems that run the same jobs continuously on the same cores. Here, performance improvements typically translate into better answers rather than finishing sooner. Systems modeled by this class of benchmarks include embedded control systems and some scientific simulations. For example, a robot might continuously run a vision analysis program on one core and a motion planning program on a second core. If an architectural change improves the vision analysis performance but not the motion planning performance, the robot will run more iterations of the vision analysis relative to the iterations of the motion planning.

E. Time-Base / Single Queue

The fourth class of benchmark is *TO* "Run for a fixed *T*ime, *O*ne Queue"). This class has a single FIFO queue, $\mathcal{F}(q, n) = \langle \mathrm{tail}(q), \mathrm{head}(q) \rangle$, and runs for a fixed time ($\mathcal{T}$ ="run for X minutes"). Figure 2 parts (a) and (b) show two different benchmarks from this class, which have similar $\mathcal{S}_0$s to the benchmarks shown in Figure 1 parts (a) and (b) respectively.

The *TO* benchmark class is similar to the *JO* class presented first. The major differences are that in *TO* the job ratio is not controlled precisely, and jobs may be partially executed when time experires. *TO*'s similarity to *JO* suggests that it is unlikely to capture any new classes of system behavior. The combination of these factors it unappealing from an experimental perspective.

F. Other Possibilities

We underscore the fact that while these four classes of benchmarks demonstrate the utility of our model, they are *not* the only types of benchmarks it can describe. One could define benchmarks with other types of state (*e.g.*, multi-level queues, a list of job/arrival time pairs, ...), or other terminating conditions (*e.g.*, end when X joules have been expended—which might be useful for research targetting mobile platforms). Experimenters should determine how to best construct benchmarks appropriate to the real-world system(s) their work targets, then describe them formally.

Furthermore, this formalism is applicable to any number of cores—not just two. As the number of cores grows, new possibilities also arise. For example, with four cores, even if one only considered queue-based definitions, new possibilites include two queues each of which is shared by a pair of cores, one core with a dedicated queue with three sharing another, and many others.

If architects desire to perform experiments where different numbers of cores are compared—*e.g.*, performance scalability from two to four to eight cores—they can still use this formalism. We note that some benchmark definitions (*e.g.*, our *JO* class) lend themselves naturally to such experiments, while others (*e.g.*, our *JM* class) are illogical in this context. Of course, as always, the experimenter should think carefully about what the real system behavior is, and define an appropriate benchmark, then formally describe it.

IV. Benchmark Definition Evaluation

Using our benchmark definition framework, benchmarks that differ in any of their four components—$\mathcal{B}$, $\mathcal{T}$, $\mathcal{F}$, or $\mathcal{S}_0$—are different, raising the question of the significance of these

	JO	*TO*	*JM*	*TM*
$\mathcal{B}$	Pairs of SPEC2006 programs run on their `train` inputs. The programs were compiled with `gcc-4.7` (Except for *dealII* which was compiled with `gcc-4.4`, as it would not compile with 4.7.) at optimization level `-O3`.			
$\mathcal{T}$	All (100) jobs	20 minutes	All (100) jobs	20 minutes
$\mathcal{F}$	As described in Section III			
$\mathcal{S}_0$ Length	100 jobs (50 of each)	∞	2 queues of 50 jobs each	∞
Ratio of programs in $\mathcal{S}_0$	1:1			—
$\mathcal{S}_0$ Order	Alternating		—	—

Table II: Details of benchmarks.

differences on experimental results. As architects generally understand that different single-program benchmarks exhibit different experimental characteristics, we focus on the multi-program-specific portions: $\mathcal{T}$, $\mathcal{F}$, or $\mathcal{S}_0$.

Typical experimental evaluations use benchmarks to compare techniques to each other. In contrast, our experimental evaluation seeks to show the importance of the benchmarks themselves. When we compare two systems, we are not trying to show which system is better; rather, we are trying to show that the benchmark definition affects the outcome of the experiment. In fact, we find that an architect can come to different conclusions regarding two systems (e.g., A is faster than B vs. B is faster than A) depending on the benchmark definition.

A. Methodology

We conduct experiments on real hardware using two systems. The first is an Intel® Sandy Bridge i7-3930k with 16GB of DRAM, and a quad-channel memory system. The second is an AMD® Bulldozer FX-8150 with 16GB of DRAM, and a dual-channel memory system. We denote the Sandy Bridge and Bulldozer as SB and BD, respectively. Both systems run Debian 7.1 with Linux kernel 3.10.10. We focus exclusively on benchmarks with two programs, so we use exactly two logical cores—the two SMT contexts on one physical core for Sandy Bridge, and the two cores in a "module" for Bulldozer—in any experiment. While our framework can be applied to benchmarks with more than two programs, we limit our evaluation to two program benchmarks in the interest of space. If anything, having more programs would make benchmark definition even more important.

For all experiments, we took many steps to minimize interference and external effects. First, all non-essential system daemons except SSH were disabled. Second, we fixed the cores' frequencies at their nominal values, disabling DVFS completely. Third, we disabled the Watchdog hang timer as well as Address Space Layout Randomization. Fourth, processes were pinned to CPUs, and memory was partitioned using Linux's NUMA emulation. Fifth, we used "real time" scheduling priority. Finally, ran each benchmark three times and used the median runtime (for *JO* and *JM*) or job execution count (for *TM*).

Our multi-program benchmarks use a variety of possible 2-program $\mathcal{B}$s, constructed from pairs of SPEC2006 single-program benchmarks. The choice of $\mathcal{B}$s is not critical to our evaluation because, unlike many experimental evaluations, we are not attempting to show that a proposed technique provides broad improvement over a representative set of benchmarks. Instead, we are showing the pitfalls and dangers of ill-defined multi-program benchmarks, so we need not concern ourselves with whether or not the $\mathcal{B}$s give comprehensive coverage; highlighting specific examples of problems is sufficient to show the need for complete definitions.

Table II shows the details of the benchmarks. While `ref` are typically used with real hardware, we used the `train` inputs to achieve reasonable execution times with large numbers of re-executions of the programs. We explore variations on some of these details in Sections IV-C and IV-D.

B. Comparison of Benchmark Classes

The top half of Figure 3 shows the results of a performance experiment using benchmarks from three of the four classes described in Section III. The fourth class, *TO*, behaves very similarly to *JO* for the reasons previously described, so we omit it in the interest of space. The graph plots the percent speedup (in terms of system throughput) of the Sandy Bridge system over the Bulldozer system.

For *JO* and *JM* benchmarks, percent speedup is quite simple—as the benchmarks hold the work constant, the percent speedup can be calculated simply as the percent reduction in runtime. For *TM*, time is held constant, so the performance improvement is in terms of increase in work done during this time. As we discussed earlier, the formula to combine the number of As and the number of Bs into on "total work" metric is system dependent. However, we are not as concerned with the particular details of that issue here (we are not trying to show one system is better than another for a specific task—instead, just showing that the benchmarks themselves behave quite different), allowing us to do anything reasonable. For *TM*, we compute the total work as the sum of the number of As executed and the number of Bs execute. We note that we have examined a variety of different metrics, and none of them change the key conclusions of this work.

The most important result from Figure 3 is that for benchmarks comprised of the same underlying single programs, constructing them in different classes produces significantly different experimental results; they are quantitatively and qualitatively different benchmarks. In fact, of the 21 single-program pairings in Figure 3, only *cactus/dealII* and *milc/bzip2* exhibit similar (within 3%) speedups across all three benchmark classes. The different results between classes mean that improperly or imprecisely defined multi-program benchmarks

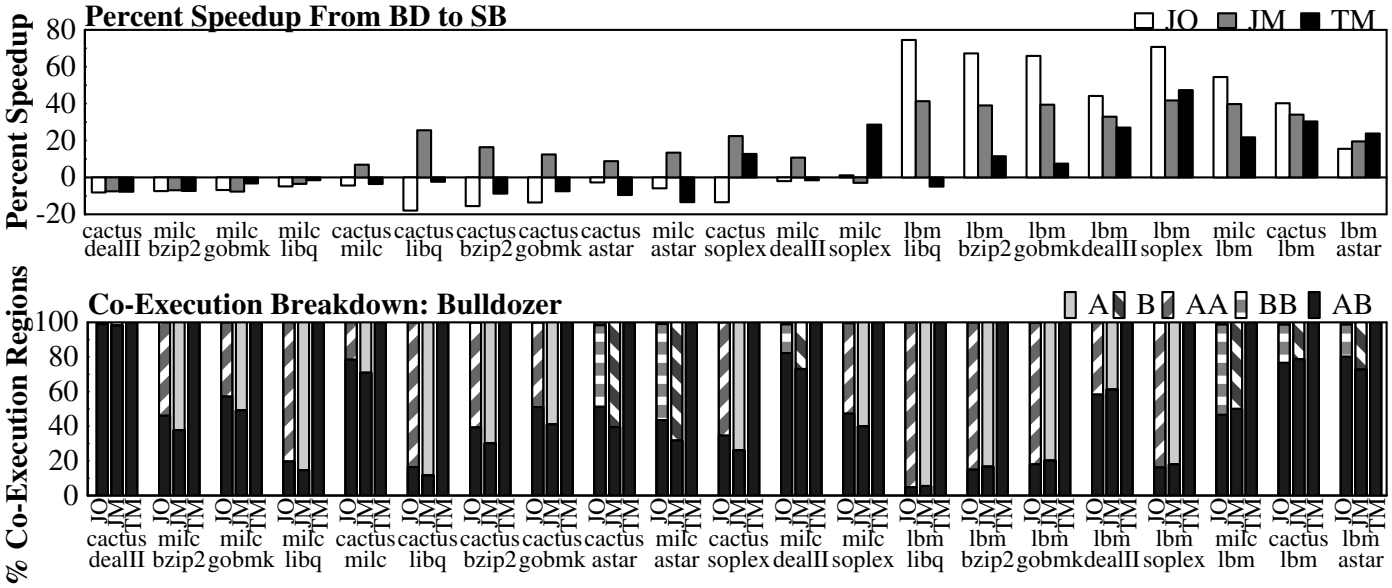

Figure 3: Top: Speedup of Sandy Bridge over Bulldozer for benchmarks from three classes. Bottom: Co-execution breakdown.

can lead to the wrong conclusion about an architectural design.

The bottom half of Figure 3 provides insights into the differences between these benchmarks by showing the *co-execution breakdown* of each benchmark on Bulldozer, *i.e.*, the percentage of execution time where different combinations of programs were executing together. The Sandy Bridge co-execution breakdown has the same high-level trends—even though the specific percentages change—so we elide it in the interest of space. Here, *AA* means that the first program in the pair executed in parallel with another instance of itself. *BB* is similar for the other program. *A* and *B* are where the respective programs executed by themselves, with the other "slot" idle (load imbalance). *AB* indicates that the two different programs co-executed. While *TM* benchmarks are 100% *AB*—by definition—the other classes exhibit different co-execution patterns, resulting in different behaviors. We now examine a few of the more interesting pairings in-depth.

cactus/dealII. One of the pairings where all three classes are similar is *cactus/dealII*. Here, the co-execution breakdowns are almost identical—more than 98% *AB* for all classes. This similarity arises because *cactus* and *dealII* have almost identical runtimes when run together.

milc/bzip2. The other pairing where all three classes are similar is *milc/bzip2*. Unlike *cactus/dealII*, the co-execution breakdowns are not similar across the three classes. Instead, this similarity is a matter of coincidence—much like one could observe similar speedups on two single-program benchmarks in a given experiment, but they would still clearly be two different benchmarks.

cactus/libquantum. The *JM* execution shows a 26% speedup for Sandy Bridge over Bulldozer, whereas the other classes exhibit slowdowns. The *JM* benchmark's behavior is dominated by load imbalance; it primarily executes *cactus* by itself, on which Sandy Bridge outperforms Bulldozer by a significant amount. The *JO* benchmark, however, exhibits an 18% slowdown. For *JO*, most of the execution is *cactus* paired with itself (as opposed to alone). The difference in memory hierarchies accounts for this difference; Bulldozer has a larger (2MB versus 256KB) L2 cache. Compared to a single-program execution, a multi-program execution of *cactus* paired with itself on Bulldozer exhibits almost no increase in L2 cache misses ($< 1\%$), whereas Sandy Bridge sees 36% more. The *TM* benchmark co-executes *cactus* with *libquantum* all the time, which Sandy Bridge executes 2% slower than Bulldozer—a different result than either of the previous two.

lbm/libquantum. Here, the *JO* and *JM* classes show significant (though significantly different) speedups of 75% and 41%. Here, *lbm* runs for significantly longer than *libquantum*. In the *JO* benchmark, this disparity results in significant *lbm*/*lbm* pairings, which Sandy Bridge executes 78% faster than Bulldozer. For *JM*, the difference between *lbm*'s and *libquantum*'s runtimes manifests as load imbalance, where one core is idle and the other executes *lbm* for most of the benchmark. The JM benchmark primarily measures the single program speedup of *lbm* run alone. Experimenting with a *TM* benchmark yields an even more drastically different answer: Sandy Bridge is 5% slower than Bulldozer. *TM*—by definition—executes *lbm* with *libquantum* the entire time, which requires it to execute significantly more *libquantum*s than *lbm*s: a 16.9:1 ratio on Sandy Bridge and a 24.4:1 ratio on Bulldozer.

C. Impact of Program Ratios in $\mathcal{S}_0$

The benchmark classes evaluated in the previous section are defined by varying $\mathcal{F}$; however, even within one class,

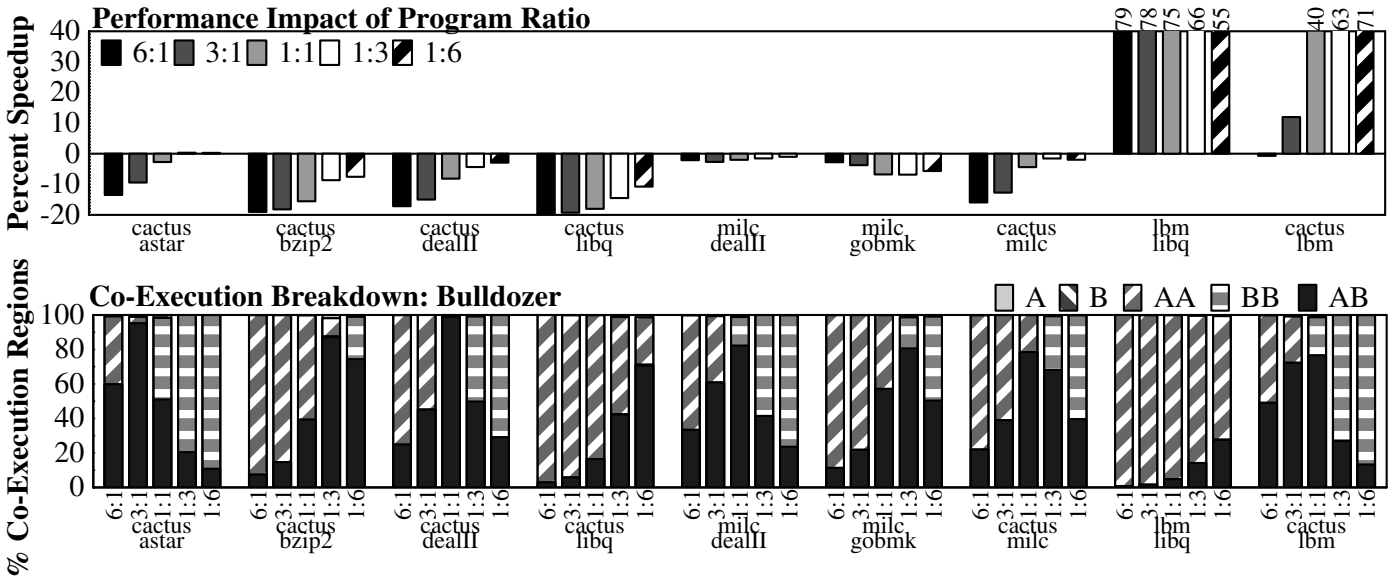

Figure 4: Top: Performance impact of varying the ratio of each program in $\mathcal{S}_0$ for *JO* benchmarks. Bottom: Corresponding co-execution breakdowns.

the specific value of $\mathcal{S}_0$ can also have significant impact on the experimental results. One characteristic of $\mathcal{S}_0$ that is particularly important to *JO* (and *TO*) benchmarks is the ratio of the two single-program benchmarks within the single queue.

The top graph in Figure 4 evaluates the impact on speedups of the ratio of the underlying single-program benchmarks in $\mathcal{S}_0$. Here, the A-to-B ratio is varied from 6A:1B (left) to 1A:6Bs (right), with 3A:1B, 1A:1B, and 1A:3B in the middle. For each benchmark, the duration is 50 of the program on the "1" side of the ratio, and an appropriate number (300, 150, or 50) of the other program. The bottom graph shows the corresponding co-execution breakdown.

The most important observation from this experiment is that the ratio of the underlying single-program benchmarks makes a significant difference. The most pronounced example is *cactus*/*lbm* where we observe a 1% slowdown at 6:1 and a 71% speedup at 1:6. Other program pairs also exhibit a significant range of results, including a mix of slowdowns and speedups depending on their ratios. Although these results are not surprising in light of the results of the prior sections, they highlight another important consideration in benchmark design; even if the other aspects of the benchmark ($\mathcal{B}$, $\mathcal{T}$, and $\mathcal{F}$) are selected properly for an experiment, the ratio between the single-program benchmarks must align with the system that the experimenter hopes to model. If the system executes the programs in a variety of ratios, altering the ratios of $\mathcal{S}_0$ is not just a minor concern, but rather definition of completely different benchmarks with distinct behaviors.

The second observation is that the performance impacts of ratio alterations do not lend themselves well to simple extrapolation. Some pairings (*e.g.*, *cactus*/*dealII*) exhibit monotonic trends, however, many other pairings (*e.g.*, *cactus*/*milc*) flatten off, or even exhibit U-shaped behavior. The exact behavior depends on the relative performance characteristics of the different co-execution phases, as well as how the co-execution breakdown differs between systems. The important takeaway is that one cannot simply extrapolate the behavior of all ratios from a few data points.

D. Impact of $\mathcal{S}_0$ Ordering

Another important consideration in the design of a *JO*-style benchmark is the ordering of the programs in $\mathcal{S}_0$. Different orderings here result in different benchmarks which can have significantly different performance characteristics. Considering the extreme case of $\mathcal{S}_0$ =ABAB... *vs.* $\mathcal{S}_0$ =AAA...BBB... yields two benchmarks that intuitively behave quite differently, even when they have the same $\mathcal{B}$, $\mathcal{F}$, $\mathcal{T}$, and A-to-B ratio in $\mathcal{S}_0$. The first $\mathcal{S}_0$, would typically exhibit a significant *AB* component in the co-execution breakdown. The second, should be exclusively *AA* and *BB*.

Figure 5 shows the impact of varying the ordering in $\mathcal{S}_0$ for benchmarks in the *JO* class. Here, we experiment with four different queue orderings, all of which maintain 50 of each program. The first ordering alternates between the two programs, which is the same ordering used in Figure 3. We denote this ordering as $(AB)^*$. The second ordering places 5 consecutive instances of program A followed by 5 consecutive instances of program B, denoted $(5A5B)^*$, and repeats this pattern. The next ordering, $(10A10B)^*$, has groups of 10. The final ordering, (50A50B), has groups of 50.

Varying the ordering of the programs in $\mathcal{S}_0$ can have noticeable impact on the performance results. For *namd*/*cactus*, alternating *namd* and *cactus* results in Sandy Bridge under-

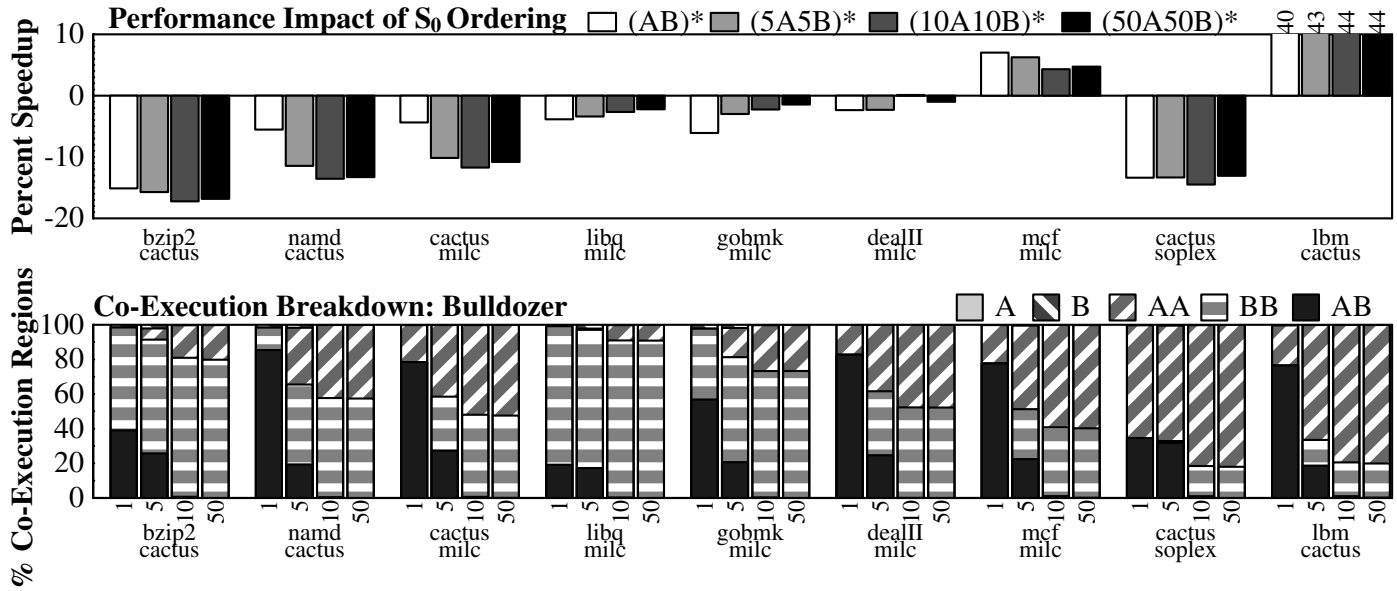

Figure 5: Top: Performance impact of program ordering in $\mathcal{S}_0$ within the *JO* class of benchmarks. Bottom: Corresponding co-execution breakdowns on Bulldozer.

performing Bulldozer by only 5.5%. However, when the programs are clustered in the queue, Bulldozer's performance advantage increases to 11.4% for (5A5B)*, and 13.3% for (50A50B). The *gobmk/milc* pairing exhibits the opposite trend—as the queue ordering becomes more clustered, Sandy Bridge performs better relative to Bulldozer. In all pairings, the more clustered ordering leads to a higher probability of *AA* and *BB* co-executions—shifting the overall performance towards those behaviors, and away from the *AB* behavior.

Interestingly, the groups-of-10 and the groups-of-50 orderings exhibit very similar—all within 1% of each other—behavior across all of the program pairings we experimented with. In fact, for any group-of-X ordering where X is even, one would expect similar behavior, as pairs of As run together (forming AA co-executions). These As then end at the same time, resulting in pairs of Bs starting and running together (forming BB co-executions). The results are not identical, as slight perturbations in the execution of one program may result in them not ending at the same time, leading to brief AB co-execution, followed by the Bs executing slightly out of phase.

Although the results in Figure 5 show that benchmarks with different $\mathcal{S}_0$ orderings have different performance behavior, the experiments suffer from the fact that a few hand-picked queue orderings were used to generate them; it is not clear how representative these orderings are. To further explore the impact of the ordering of the programs in $\mathcal{S}_0$, Figure 6 shows the PDFs of speedups for 150 randomly selected orderings of $\mathcal{S}_0$ on each of four program pairings, as well as where the (AB)* pairing falls on that distribution.

There are two interesting observations to make from Figure 6. First, the distributions are basically Gaussian, but with

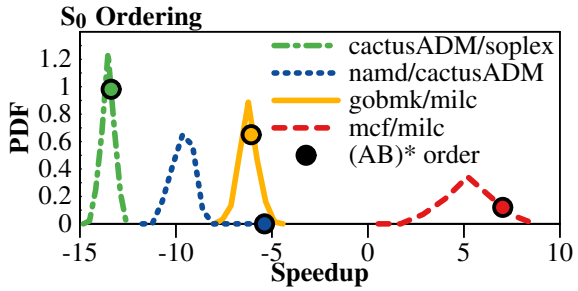

Figure 6: Variation in performance for 150 uniformly randomly chosen $\mathcal{S}_0$ orderings for 4 program pairings.

rather different variances for the program pairings. The most narrow distribution of these three is *cactus/soplex*, with a standard deviation (σ) of 0.32 percent speedup. This standard deviation means we would expect to find 99.7% of all speedups within $\pm 3\sigma = 0.96$ percent of the mean—less than a 2% range. By contrast, *mcf/milc* has a much wider distribution, with $\sigma = 1.16$. Here, the $\pm 3\sigma$ range spans almost 7% speedup.

The second interesting observation from this data is that the (AB)* orderings used in the previous experiments (shown by the circles) fall at different points in the distribution—for some pairings (*e.g.*, *cactus/soplex*), the speedup obtained with the (AB)* ordering is quite close to the mean. By contrast, *namd/cactus*'s (AB)* results are well outside the randomly generated results.

The important take-away point from this experiment is that $\mathcal{S}_0$ ordering is a significant experimental concern. If an experimenter cannot say with certainty what $\mathcal{S}_0$ accurately models reality, multi-program results for *JO*-style benchmarks

are only meaningful if they evaluate multiple $\mathcal{S}_0$ orderings which align with the orderings that occur in the real world situations they intend to model.

V. Related Work

Prior work has explored aspects of multi-program benchmarking, but no prior work has precisely and clearly defined multi-program benchmarks as we have.

A. Multi-program Benchmark Definition

FAME [21] constructs multi-program benchmarks by repeating the executions of the single programs enough times to ensure that a steady-state behavior is achieved. In our benchmark definition terminology, FAME proposes setting $\mathcal{T}$ to be "steady state behavior is obtained." The FAME work does not consider the $\mathcal{F}$ or $\mathcal{S}_0$ aspects of benchmark definition. In those regards, the FAME benchmarks are related to the *TM* class of benchmarks that we have proposed here, but without the benefit of precisely defined $\mathcal{F}$ or $\mathcal{S}_0$.

Other work [20] examines the methods for selecting the single-program benchmark pairings—in our terminology, $\mathcal{B}$—from which to define representative workloads. Selecting appropriate values of $\mathcal{B}$ is important, but it is orthogonal to our work.

B. Sampling Multi-program Benchmarks

Much prior multi-program work has explored *sampling* methodologies [5], [11], [13], [18], [19], [22]. This work explores how to select sub-sets of multi-program benchmarks to execute so as to obtain representative results. This work on sampling has focused on the sampling aspect rather than on the benchmarks from which they are sampling. Instead of formalizing the entire benchmarks that the samples represent, which is our focus, this prior work has let the sampling methodologies implicitly define the complete benchmarks—the benchmark is whatever the samples happen to represent.

Variable Instruction Count Sampling. A variety of sampling methodologies define their samples by executing both programs together until some condition is reached, then terminating both programs. These methodologies are often referred to as *variable instruction count* because they allow architectural differences to result in changes in the number of instructions executed from each program. Approaches to variable instruction count sampling include "run both programs, possibly starting from checkpoints, until the sum of their committed instruction counts reaches a specified target" (*e.g.*, [16]), "run until both programs reach a minimum number of instructions or until one program reaches a maximum number of completed instructions" (*e.g.* [11]), and "run until each program has reached a minimum number of instructions, restarting the faster program as needed" (*i.e.,* FAME [21]). The samples from these methodologies represent benchmarks similar to those in the *TM* category; the ratios of their constituent programs vary depending on the underlying micro-architecture.

A notable approach to variable instruction count sampling is the Co-Phase Matrix methodology [19]. This approach is based on determining the behavior of two programs in a *co-phase*: a period during which their behaviors and contention exhibit a consistent pattern. Assuming two programs paired together, such that the first program has M phases and the second has N phases, one can construct a $M \times N$ co-phase matrix that represents the performance behavior of all co-phases of the two programs. For more than two programs, the dimensionality of the co-phase matrix increases; K programs form a K-dimensional matrix. Each entry in the co-phase matrix is the performance of that co-phase, and it is obtained with detailed simulation. The overall behavior of the multi-program execution is then estimated by a fast analytical simulation which tracks what phase each program is in, looks up the performance characteristics of that co-phase in the co-phase matrix, and then determines how long the co-phase will last (*i.e.*, how long until either program changes phase behavior). The process repeats until the end of the sample of one program is reached, which means this approach is a variable instruction count methodology. Later work describes how to use a cluster analysis (somewhat similar to SimPoint [14]) to determine representative co-phases of programs [18].

While Co-Phase Matrix, as proposed, implicitly samples from *TM*-style benchmarks, we observe that it can be adapted to sample any of our proposed benchmark classes. First, for single-queue benchmark classes (*JO* and *TO*), the co-phase matrix for programs A and B may need to be expanded to include A/A and B/B pairings, whereas the original co-phase matrix needs only A/B pairings. Second, the analytical simulation needs to be extended to track the state of the queue(s), and to "run" (selecting elements from $\mathcal{B}$ to assign to cores according to $\mathcal{F}$) until $\mathcal{T}$ indicates termination. We note that extending co-phase matrix is not limited to the queue-based classes we proposed, but should be feasible for *any* multi-program benchmark.

Fixed Instruction Count Sampling. Other sampling methodologies (*e.g.*, FIESTA [5]) construct their samples to ensure that the same subsets of the dynamic instruction stream will be executed regardless of micro-architectural differences. Accordingly, these methodologies are called *fixed instruction count*, because they ensure that the instruction counts from each program (and thus ratios) remain fixed. These methodologies mostly correspond to benchmarks in the *JM* category, however they may do a poor job of modeling the load-imbalance behavior in *JM* benchmarks. Specifically, an actual *JM* benchmark will experience all of its load imbalance at the end; one program may execute (possibly for multiple iterations) by itself while the other core remains idle. However, these sampling methodologies experience load imbalance at the end of each sampling epoch. The load imbalance experienced during the sampling may not correspond to the load imbalance in the full execution; in fact, it may not even have the correct program running longer. FIESTA [5] attempts to

reduce *sample imbalance*—load imbalance arising from constructing samples with uneven stand-alone execution time—which is appropriate to the specific subset of *JM* benchmarks where $\mathcal{S}_0$ has a program ratio balanced to match stand-alone execution times.

VI. What Should Experimenters Do?

Having seen the impact of benchmark definition on experimental results, the natural question to ask is "So what should an experimenter do to perform multi-program experiments that are meaningful and reproducible?" We advise experimenters to follow the steps outlined in Section II while making choices at each step that are sensible for the expected use of the system being evaluated. For Step 1 (benchmark selection), the experimenter must choose appropriate benchmark programs for the target use case. For Step 2 (benchmark definition), which is the focus of this paper, the experimenter must precisely define the multi-program benchmark in a way that is appropriate for the target use case. For Step 3 (metric selection), the experimenter must choose metrics that are appropriate for the target use case. For Step 4 (benchmark sampling), the key is to first define the complete benchmark (in Step 2) and **then** choose a statistically sound sampling methodology. This 4-step procedure is intuitive and perhaps obvious (at least in retrospect), yet it is not the standard practice in the field.

Another possible response to this paper is a desire for a single universal multi-program benchmark definition. However, as we have stated before, there is no single universal answer of this form. We have posed three benchmark classes—*JO*, *JM*, and *TM*—which each represent different characteristics, modelling different real system behaviors. An experimenter may find one of these classes appropriate to one research idea. For another experiment, the research may find none of these classes to be appropriate in which case, our formalism provides the experimenter with the tools to precisely define and describe the benchmarks they need.

VII. Acknowledgments

This material is based on work supported by the National Science Foundation under grant CCF-125-9028.

References

[1] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2013, pp. 77–88.

[2] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, 2008.

[3] ——, "Restating the case for weighted-ipc metrics to evaluate multiprogram workload performance," *IEEE Computer Architecture Letters*, vol. 99, no. 2, p. 1, 2013.

[4] A. Hilton, N. Eswaran, and A. Roth, "CPROB: Checkpoint Processing with Opportunistic Minimal Recovery," in *Proc. 18th Intl Conf on Parallel Architectures and Compilation Techniques*, Sep. 2009.

[5] ——, "FIESTA: a sample-balanced multi-program workload methodology," in *Third Annual Workshop on Modeling, Benchmarking, and Simulation (MoBS)*, San Diego, CA, USA, 2007.

[6] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez, "Balancing dram locality and parallelism in shared memory cmp systems," in *18th International Sypmosium on High Performance Computer Architecture (HPCA)*. IEEE, 2012, pp. 1–12.

[7] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM journal of Research and Development*, vol. 49, no. 4.5, pp. 589–604, 2005.

[8] P. Michaud, "Demystifying multicore throughput metrics," *IEEE Computer Architecture Letters*, vol. 12, no. 2, p. 63, 2012.

[9] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith, "Fair queuing memory systems," in *39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2006, pp. 208–222.

[10] M. Pericas, A. Cristal, F. J. Cazorla, R. Gonzalez, D. A. Jimenez, and M. Valero, "A flexible heterogeneous multi-core architecture," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. IEEE Computer Society, 2007, pp. 13–24.

[11] S. Raasch and S. Reinhardt, "The impact of resource partitioning on SMT processors," in *12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2003, pp. 15–25.

[12] N. Rafique, W.-T. Lim, and M. Thottethodi, "Effective management of dram bandwidth in multicore processors," in *Parallel Architecture and Compilation Techniques, 2007*. IEEE, 2007, pp. 245–258.

[13] Y. Sazeides and T. Juan, "How to compare the performance of two SMT microarchitectures," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2001, pp. 180–183.

[14] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[15] A. Snavely and D. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreading Processor," in *Proc. 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2000, pp. 234–244.

[16] A. Snavely, D. M. Tullsen, and G. Voelker, "Symbiotic jobscheduling with priorities for a simultaneous multithreading processor," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 30, no. 1. ACM, 2002, pp. 66–76.

[17] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "Mise: Providing performance predictability and improving fairness in shared main memory systems," in *19th International Symposium on High Performance Computer Architecture (HPCA 2013)*. IEEE, 2013, pp. 639–650.

[18] M. Van Biesbrouck, L. Eeckhout, and B. Calder, "Representative multiprogram workloads for multithreaded processor simulation," in *IEEE 10th International Symposium on Workload Characterization (IISWC)*, 2007, pp. 193–203.

[19] M. Van Biesbrouck, T. Sherwood, and B. Calder, "A co-phase matrix to guide simultaneous multithreading simulation," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2004, pp. 45–56.

[20] R. A. Velásquez, P. Michaud, and A. Seznec, "Selecting benchmarks combinations for the evaluation of multicore throughput," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013.

[21] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero, "Fame: Fairly measuring multithreaded architectures," in *16th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2007, pp. 305–316.

[22] T. Wenisch, R. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. Hoe, "SimFlex: statistical sampling of computer system simulation," *IEEE Micro*, vol. 26, no. 4, pp. 18–31, 2006.