Appears in Proceedings of 39th International Symposium on Microarchitecture (MICRO-39), Dec. 11-13, 2006.

# Serialization-Aware Mini-Graphs: Performance with Fewer Resources

Anne Bracy* and Amir Roth
Department of Computer and Information Science, University of Pennsylvania
*{bracy, amir}@cis.upenn.edu*

## Abstract

*Instruction aggregation—the grouping of multiple operations into a single processing unit—is a technique that has recently been used to amplify the bandwidth and capacity of critical processor structures. This amplification can be used to improve IPC or to maintain IPC while reducing physical resources. Mini-graph processing is a particular instruction aggregation technique that targets dynamically-scheduled superscalar processors and achieves bandwidth and capacity amplification throughout the pipeline.*

*The dark side of aggregation is serialization. External serialization is an effect common to many aggregation schemes. An aggregate cannot issue until all of its external inputs are ready. If the last-arriving input to an aggregate feeds what is not the first instruction, the entire aggregate can be delayed. Mini-graphs additionally suffer from internal serialization. Serialization can degrade performance, sometimes to the point of overwhelming the benefits of aggregation.*

*This paper examines the problem of serialization and serialization-aware aggregation in the context of mini-graphs. An aggressive mini-graph selection scheme that seeks to maximize amplification, produces amplification rates of 38% but, due to serialization, cannot use them to compensate for a 33% reduction in physical resources (i.e., a reduction from 4-way issue to 3-way issue). A conservative selection scheme that avoids serialization by static inspection produces amplification rates of only 20%, making a performance neutral reduction in resources virtually impossible.*

*To reconcile the seemingly conflicting goals of resource amplification and serialization avoidance, this paper develops three schemes that identify and reject mini-graphs with harmful serialization. The most effective of these, Slack-Profile, uses local slack profiles to reject mini-graphs whose estimated delay cannot be absorbed by the rest of the program. Slack-Profile virtually eliminates serialization-induced slowdowns while providing 34% amplification rates. A 3-way issue processor augmented with Slack-Profile mini-graphs outperforms a 4-way issue processor by an average of 2%.*

## 1. Introduction

Instruction aggregation—the grouping of multiple operations into a single processing unit—is a technique that has received significant attention in recent years. Traditionally, aggregation techniques have targeted performance, using custom functional units to reduce aggregate execution latency [5, 18]. More recently, aggregation has been used as a "complexity-effective" superscalar performance technique, amplifying the effective bandwidth and capacity of critical structures like the issue queue. By performing certain actions once per aggregate instead of on a per-instruction basis [1, 11, 19], structure bandwidth and capacity can be allocated to other instructions, creating an amplification effect. ***Mini-graph processing*** is a form of instruction aggregation that targets performance efficiency via bandwidth and capacity amplification throughout the entire pipeline, from instruction cache to commit. This wholesale amplification enables either improved IPC throughput at a fixed resource point or, alternatively, fixed (or better) IPC with fewer resources.

Instruction aggregation can improve performance either directly (via custom acceleration) or indirectly (via resource amplification). However, it also has the potential for direct performance degradation. The mechanism for this is ***serialization***, and there are two forms. ***External serialization***, the more frequent and destructive form, occurs because an aggregate cannot issue until all of its external register inputs are available. Delay can result when the last-arriving input to the aggregate is not an input to the first instruction. ***Internal serialization***, a less dominant form, occurs if the technique requires aggregate constituent instructions to execute in series even when those instructions may be independent. Effectively exploiting instruction aggregation requires avoiding serialization penalties.

This paper examines the problem of intelligent, serialization-aware instruction aggregation in the context of mini-graph processing. Mini-graphs are vulnerable to both forms of serialization, although most aggregation schemes are vulnerable to external serialization, and many are also vulnerable to internal serialization as well [15, 18]. Mini-graphs also suffer

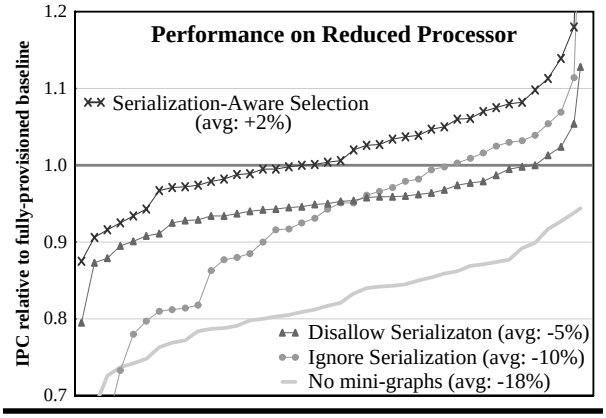



**FIGURE 1. Serialization-aware Mini-graph Selection.** Performance on a reduced processor relative to a fully provisioned one for 78 programs. The serialization-aware *Slack-Profile* (X) mini-graph selector outperforms two naive selectors and (on average) allows mini-graphs to compensate for the performance loss of the reduced configuration (grey line).

* Anne Bracy is also a member of the Microarchitecture Research Lab (Microprocessor Technology Labs, Corporate Technology Group) of Intel Corporation.

from serialization more than most other aggregation techniques because they don't provide a direct latency-reduction benefit that can directly counter-act serialization-induced delay. Finally, by focusing on resource amplification, mini-graphs require wholesale (rather than opportunistic) aggregation, increasing the probability of harmful aggregates.

In mini-graph processing, resource amplification is proportional to ***dynamic coverage***, the fraction of dynamic instructions "embedded" in mini-graphs. ***Struct-All***, a naive, serialization-blind mini-graph selection algorithm that attempts to maximize amplification achieves average coverage rates of 38%. Without serialization, these amplification rates should allow mini-graphs to compensate—in terms of performance—for a reduction from 4-way fetch/issue/commit to 3-way fetch/issue/commit and from 30 issue queue entries and 80 rename registers to 20 and 56, respectively; a reduction that typically results in an 18% slowdown, on average. Unfortunately, *Struct-All* admits a small number of pathological serializing mini-graphs that degrade the performance of 40% of all programs, even on the fully-provisioned processor. ***Struct-None***, a selection algorithm that conservatively rejects any mini-graph whose dataflow structure makes it vulnerable to serialization, eliminates these slowdowns but also cuts coverage to 20%, making it impossible to achieve performance neutrality on the reduced processor.

The problem with *Struct-None* is that serialization and serialization-induced performance loss cannot be deduced by inspecting dataflow structure. Some mini-graphs are structurally vulnerable to serialization but do not manifest it at runtime, because the potentially serializing input is always ready first. Other mini-graphs manifest serialization but don't degrade performance because the delayed output is "off the critical path." This paper reconciles the tension between maximizing coverage and minimizing serialization penalties by presenting three mini-graph selection algorithms that are serialization-aware in a more sophisticated way.

- ***Struct-Bounded*** uses program structure to accept aggregates whose delay can be qualitatively bounded.
- ***Slack-Profile*** uses local slack profiles [7] to quantify the delay induced by mini-graph formation, to estimate whether that delay can be absorbed by the rest of the program, and to reject mini-graphs whose estimated delay cannot be absorbed.
- ***Slack-Dynamic*** is a hardware implementation of *Slack-Profile*. It monitors actual execution to identify and disable mini-graphs that actually suffer from serialization delay and whose delay is actually propagated to consumers.

Cycle-level simulation of benchmarks from four suites shows that *Slack-Profile* successfully reconciles the competing goals of high coverage and serialization avoidance. It produces average coverage rates of 34% and allows the mini-graph enabled reduced processor to actually out-perform its fully-provisioned counter-part by 2% (see Figure 1). It also shows that the profile information on which *Slack-Profile* is based is robust to both gross microarchitectural features and program input data sets. *Struct-Bounded* and *Slack-Dynamic* are somewhat less effective, managing to reduce performance loss to only 2% and 6% (down from 18%), respectively, but require no profiling information. All three selectors are compatible with aggregate techniques beyond mini-graph processing.

The next section reviews the basics of mini-graph processing. Section 3 evaluates serialization-unaware selection and motivates the new serialization-aware selectors. *Struct-Bounded*, *Slack-Profile*, and *Slack-Dynamic* are presented in Section 4 and evaluated in Section 5. Section 6 discusses related works and other aggregate selection techniques.

## 2. Mini-Graph Primer

Mini-graphs are instruction aggregates that are specifically tailored to exploit capacity and bandwidth amplification in dynamically scheduled superscalar processors. This section reviews the basics of mini-graphs, mini-graph selection, and mini-graph processing.

**Definition.** Mini-graphs are aggregates with the external interfaces of singleton RISC instructions: they are atomic units with a maximum of three register inputs, one register output, one memory reference, and one control transfer [1].

The RISC singleton interface makes mini-graphs appropriate for superscalar processors which rely on simple bookkeeping units to implement register renaming and dynamic scheduling (this is the function of micro-ops, after all). Atomicity is the key to amplification, it allows register communication that is "interior" to a mini-graph to take place without actual registers and thus to amplify both the capacity of the physical register file as well as the bandwidths of all stages that manipulate either register names or values. Atomicity restricts mini-graphs to basic blocks.

As originally described, mini-graphs were limited to two register inputs and internal dataflow connectivity [1]. To boost amplification, this work relaxes these constraints. Support for a third register input includes an additional register tag and match bus in the issue queue, and one or two additional register file read ports. These changes are acceptable considering mini-graphs otherwise amplify issue queue and register file capacity and bandwidth. Internally disconnected mini-graphs require no special support.

**Encoding.** The need to identify mini-graph "interior" register values requires liveness analysis [3] and implies static identification. A software tool (compiler or binary rewriter) identifies instruction groups that satisfy mini-graph criteria and encodes them into the executable.

Mini-graphs use a new encoding scheme called "outlining" [17], that supports functional compatibility on non mini-graph processors and enables instruction cache capacity and fetch bandwidth amplification on mini-graph processors [2]. The instructions that form the body of the mini-graph are prepended with a new instruction that includes a special opcode, the names of the mini-graph's interface register, and a short identifier. This extended sequence is then "outlined" from the code (as opposed to "inlined") using a pair of jumps (Figure 2b). On a non-mini-graph processor, the special instruction that precedes the mini-graph is a nop; the processor jumps to the outlined location, ignores the nop, executes the mini-graph instructions, and jumps back to the program main line. On a mini-graph processor, the instruction cache fill path recognizes the special instruction and performs the following transformation (Figure 2c). In the instruction cache, the outlining jump (*i.e.*, the jump to the mini-graph) is replaced by the

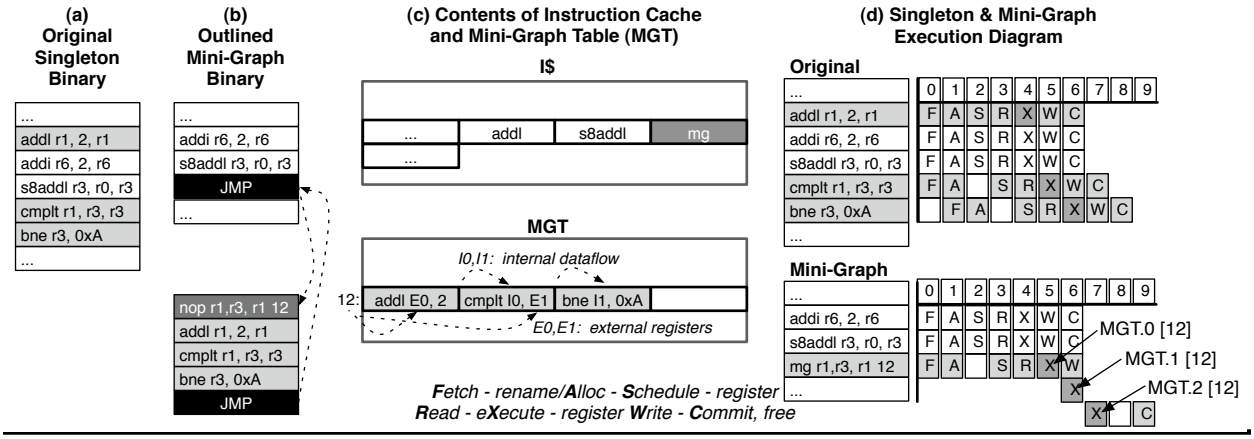



**FIGURE 2. Mini-graph Basics. (a)** Original singleton binary. **(b)** Outlined mini-graph binary: the mini-graph is removed from its original static location and replaced with a jump. **(c)** Contents of instruction cache and Mini-Graph Table (MGT). Instruction fill unit places outlined code into MGT, overwrites the outlining jump with the handle in the I$. **(d)** Singleton and mini-graph execution on a 4-way fetch/issue/commit pipeline. In mini-graph execution, the handle is processed as a singleton at all stages but execute. Mini-graph execution consumes less bandwidth and fewer issue queue slots and physical registers.

special instruction, which acts as the mini-graph's ***handle***. The mini-graph constituent instructions are pre-processed and written to the ***mini-graph table (MGT)***, an on-chip cache that holds mini-graph template descriptions (*i.e.*, constituent operations and their dataflow), at an index corresponding to the short identifier in the handle.

**Execution.** A mini-graph processor fetches handles and treats them as singleton instructions at every pipeline stage except execute. The scheduler, the only stage modified to recognize handles, invokes the MGT, which then drives the cycle-by-cycle execution of the mini-graph's constituent instructions, micro-code style.

Mini-graphs naturally amplify the capacity of all book-keeping structures and the bandwidths of all book-keeping stages. To prevent execution itself from becoming a bandwidth bottleneck, a mini-graph processor replaces some of its ALUs with ***ALU pipelines***, single-entry, single-exit chains of ALUs with forward-only interior operand networks. ALU pipelines add ALU execution bandwidth without requiring matching increases in register file and bypass bandwidths.

Figure 2d shows an example of singleton (*i.e.*, non-mini-graph) and mini-graph execution on a 4-way issue pipeline. Whereas singleton execution consumes five fetch, rename, schedule, writeback and retire slots, mini-graph execution consumes only three.

**Selection.** The goal of a mini-graph selection algorithm is to maximize dynamic coverage (resource amplification) given: (i) an initial pool of static mini-graph candidates, and (ii) a static template budget, *i.e.*, the size of the MGT.

The initial pool of mini-graph candidates is formed by some combination of static analysis, heuristics, and profiling information. All selectors start with the same pool of mini-graphs that are not vulnerable to serialization by virtue of their dataflow shape. However, different selectors start with different pools of potentially-serializing mini-graphs, *i.e.*, mini-graph that have an external register input to any instruction other than the first. *Struct-All*, an aggressive selector, admits all potentially serializing mini-graphs into the starting pool. *Struct-None*, a conservative selector, admits none of them. A more refined selector may use some profiling information or heuristics to obtain a starting pool that includes some, but not all, potentially-serializing mini-graphs.

Once the starting pool is set, all selectors follow the same basic procedure. First, mini-graph candidates from multiple static locations that can share an MGT template are grouped. Each template is assigned a coverage score which is computed as $(n–1)*f$, where $n$ is the mini-graph's size in instructions and $f$ is the (estimated or profiled) dynamic execution frequency of all instances of the template. The selection algorithm then iteratively chooses the template with the highest score and discounts the scores of remaining templates whose instances overlap with instances of the chosen template (mini-graphs must be dynamically disjoint and so selecting one mini-graph precludes the selection of any overlapping mini-graph). Iteration terminates when the template budget is reached.

## 3. Evaluation I: Serialization-Blind Selection

To motivate serialization-aware mini-graph selection, this section presents a short evaluation and analysis of *Struct-All* and *Struct-None*.

### 3.1. Methodology

We use the 78 benchmarks from the SPECint2000, MediaBench [16], CommBench [21], and MiBench [9] suites, compiled for the Alpha EV6 using the Digital OSF compiler with optimization flags –O3. All benchmarks were run to completion: SPEC programs on their training inputs at 2% periodic sampling with warm-up; all other benchmarks on their largest available inputs with no sampling. Not all of these suites (*e.g.*, MiBench) actually target dynamically scheduled superscalar processors. We include them to show the applicability of aggregation and mini-graphs to different kinds of codes.

The timing simulator uses the SimpleScalar 3.0 Alpha AXP ISA and system call modules to model a dynamically scheduled superscalar processor. Table 1 details both the fully-provisioned and reduced configurations, and the mini-graph support. The fully-provisioned baseline is tuned to the performance "knee" for both issue queue entries (30) and physical

| Parameter | Configuration |
|---|---|
| Memory System | 32KB, 2-way associative 3-cycle access instruction and data caches. 64-entry, 4-way associative instruction and data TLBs. 1MB, 4-way associative, 12-cycle access on-chip L2. Infinite, 200 cycle-access main memory. 16B memory bus clocked at 1/4 core frequency. |
| Branch Prediction | 24Kb hybrid bimodal/gShare branch direction predictor, 2K-entry, 4-way associative BTB, 32-entry RAS |
| Pipeline | 13 stages: 1 predict, 3 I$, 1 decode, 2 rename, 1 schedule, 2 regread, 1 execute, 1 regwrite, 1 commit |
| Instruction Window | 128-entry ROB, 48-entry load queue, 32-entry store queue. Loads are scheduled aggressively using a 1K-entry StoreSets predictor. Memory ordering violations flush the pipeline. Cache miss replays are modeled. |
| Baseline Processor | 4-way fetch/issue/commit, 30-entry issue queue, 144 physical registers. The scheduler may issue up to 4 simple integer, 1 complex integer/floating-point, 2 loads and 1 store per cycle. |
| Reduced Processor | 3-way fetch/issue/commit, 20-entry issue queue, 120 physical registers. The scheduler may issue up to 3 simple integer, 1 complex integer/floating-point, 1 load and 1 store per cycle. |
| Mini-Graphs | Mini-graphs are up to 4 instructions long. The scheduler issues at most 2 mini-graphs per cycle, only one of which may contain a memory operation. 512-entry MGT. 2 4-stage ALU pipelines. |

**TABLE 1. Simulated Processors.** For fully-provisioned and reduced processors. Shared configuration aspects are in white.

registers (144). A configuration with 40 issue queue entries and 164 registers outperforms this baseline by only 1.5%.

Most of the data is displayed using S-curve graphs. Each line represents an experiment in which all programs are sorted from worst to best; hashes mark every other program. In the same graph, each experiment is sorted independently so that the same horizontal point may correspond to different programs in different experiments. S-curves effectively display trends and medians for large numbers of benchmarks and prevent outliers from hiding in averages.

### 3.2. Performance and Coverage

The top graph in Figure 3 shows performance—IPC relative to the fully-provisioned processor, whose own performance corresponds to the y=1 axis—for mini-graphs selected by the *Struct-All* and *Struct-None* schemes. The reduced processor alone (*no mini-graphs*, light grey line) is on average 18% slower than the fully-provisioned baseline. To provide additional insight, the bottom graph shows mini-graph performance on the fully-provisioned processor. Here, any performance loss can be attributed to mini-graph serialization.

**Struct-All.** On the reduced configuration, *Struct-All* mini-graphs (circle) compensate for half of the original 18% performance loss, yielding an average slowdown of 10% relative to the fully-provisioned baseline. However, individual results vary greatly. For some programs (on the right side of the graph), *Struct-All* allows the reduced processor to out-perform the baseline. On 7 programs, it yields lower performance than the reduced processor with no mini-graphs at all. Actually, *Struct-All* produces performance degrading mini-graph sets on 29 programs. For 22 of these, however, this performance degradation is hidden on the reduced processor, which translates amplification to performance at a high rate. On the fully provisioned processor, on which amplification provides fewer performance benefits and serialization penalties are more exposed, these slowdowns are apparent.

**Struct-None.** On the reduced configuration, *Struct-None* mini-graphs (triangle) produce better average performance, compensating for 13% of the original 18% performance loss. Performance gains are also more consistent. *Struct-None* always outperforms the non-mini-graph processor. However, for about half the programs, it provides less performance than *Struct-All*. The key here is coverage. *Struct-All* yields coverage rates from 18% to 60% (38% on average). By conservatively rejecting all mini-graphs with serialization potential, *Struct-None* has only half this coverage, ranging from 6% to 38% (20% on average). On the fully provisioned processor, where amplification provides relatively less benefit and serialization is relatively more costly, this strategy allows *Struct-None* to consistently outperform *Struct-All*. However, on the reduced processor, where coverage's importance is increased and seri-

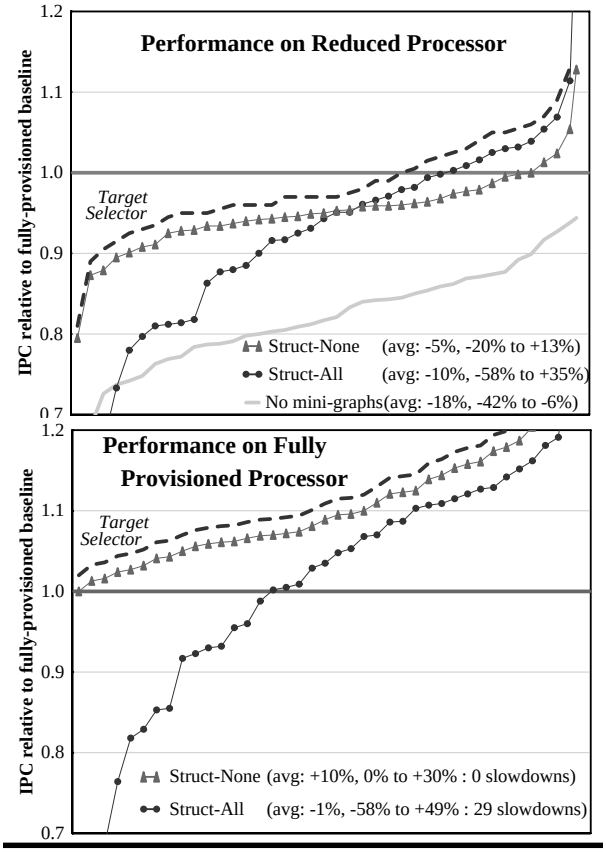



**FIGURE 3. Naive Structural Selectors. Top:** Performance on the reduced processor. **Bottom:** Performance on the fully provisioned processor. *Struct-All* (circle) accepts all serialization; *Struct-None* (triangle) accepts none. Target Selector (dashed) is strictly better than both.

alization's decreased, *Struct-None*'s performance is "shifted down" relative to *Struct-All*. Here, *Struct-All*'s superior coverage allows it to outperform *Struct-None* for about half of the programs.

**The case for a serialization-aware "hybrid" scheme.** The different shapes of the *Struct-All* and *Struct-None* S-Curves and their "cross-over" behavior on the reduced processor illustrate the tension between resource amplification on one hand and serialization on the other. They also suggest the existence of an intelligent hybrid scheme. Any hybrid should provide the "best of either world", matching the performance of *Struct-All* when amplification is at a premium and *Struct-None* when amplification is ineffective and serialization dominates. However, a hybrid scheme that is intelligently serialization-aware should be able to consistently outperform both *Struct-None* and *Struct-All*, as it should be able to make coverage vs. serialization decisions on a per mini-graph basis. The expected performance of this hybrid (*Target selector*, dashed black line) is sketched on each graph.

## 4. Serialization-Aware Mini-Graph Selection

This section first discusses the problem of serialization and then presents three serialization-aware mini-graph selectors: the heuristic *Struct-Bounded* and the quantitative *Slack-Profile* and *Slack-Dynamic*.

### 4.1. What Exactly is Serialization?

A dynamically scheduled processor ostensibly executes singleton instructions in data dependence order. An aggregation-induced serialization is an artificial dependence between two singleton instructions that is created when both instructions are placed in the same aggregate.

In the context of instruction aggregation, there are two forms of serialization. ***External serialization*** is a dependence of the first instruction in an aggregate on any instruction outside the aggregate that produces a value for any instruction in the aggregate other than the first. External serialization occurs because an aggregate cannot issue until all of its external register inputs are available. Most forms of instruction aggregation that we know of are subject to external serialization. ***Internal serialization*** is a dependence between an instruction in an aggregate and a previous independent instruction in the same aggregate. Internal serialization occurs only in techniques in which aggregate constituents must execute in series. Most aggregation techniques (including mini-graphs) are subject to internal serialization as well.

Figure 4 shows an example of serialization using six abstract instructions, A-F. Figure 4a shows a singleton execution of these instructions (assume microarchitectural constraints create an execution schedule that is longer than the height of the dataflow graph). Figure 4b shows the execution of the same instructions, but with instructions C, D, and E aggregated into a mini-graph. External serialization effectively creates a new dependence edge between instructions B and C. Internal serialization effectively creates an edge between instructions C and D. Notice, each serializing edge induces a 1-cycle delay on the corresponding instruction: C is delayed by 1 cycle, as is D. However, the total delay on the output of the aggregate, E, is 1 cycle. Here, the external serialization

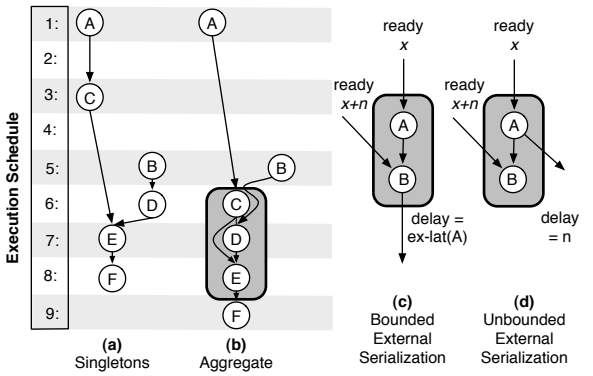



**FIGURE 4. Serialization. (a)** Singleton execution. **(b)** Aggregate execution and resulting serialization. **(c)** Bounded serialization. **(d)** Unbounded serialization.

*masks* the internal serialization.

Our experience with mini-graphs shows that internal serialization is often masked by external serialization. This suggests that, at least for mini-graphs, an aggregate execution model that simplifies implementation at the cost of adding internal serialization is a reasonable design choice.

### 4.2. Struct-Bounded

*Struct-All* and *Struct-None* are two extreme structural approaches to dealing with potentially serializing mini-graphs. *Struct-Bounded* represents a heuristic compromise. The observation behind *Struct-Bounded* is that a dynamically scheduled execution core can tolerate short execution delays (*e.g.*, data cache misses) quite well, but is less effective at tolerating longer delays (*e.g.*, L2 misses). In line with this reasoning, *Struct-Bounded* accepts mini-graphs whose serialization-induced delay can be bounded (*i.e.*, proven to be short) by inspection and rejects only ones with "unbounded" delay.

A mini-graph has a single register output, but may also have a memory output (via a store) and a control output (via a branch). From the point of view of dynamic scheduling, stores act as outputs only if they forward their values to younger in-flight loads (empirically, most stores do not) and branches only act as outputs when they are mis-predicted (empirically, most branches are not). Without knowledge of which stores forward and which branches mis-predict, *Struct-Bounded* attempts to bound the delay on a mini-graph's register output only.

Figure 4 uses abstract examples to illustrate bounded and unbounded serialization delays. Bounded serialization is any serialization that delays a mini-graph's register output by a number of cycles that is less than the execution latency of the entire mini-graph. Clearly, all strictly internal serialization is bounded. Just as clearly, all disconnected mini-graphs are prone to unbounded serialization; if the input to the instruction that does not produce the mini-graph's output arrives *n* cycles after the input to the instruction that does, the mini-graph's output is delayed by *n* cycles. In fully connected mini-graphs with external serialization, serialization delay is bounded if the serializing input is "upstream" from the register output. The mini-graph in Figure 4c has bounded serialization. Even if this serializing input is ready *n* cycles after the input to the first instruction in the mini-graph, the delay on the mini-graph's

output (B) would be equal only to the latency of instruction A. This is because in a singleton execution, B would wait for the serializing input anyway. In contrast, the slightly different mini-graph in Figure 4d is vulnerable to unbounded serialization. Here the serializing input is "downstream" from the mini-graph register output. If the serializing input is ready *n* cycles after the input to A, the mini-graph's output is delayed by *n* cycles. Here, in a singleton execution, A would not wait for B's input.

In general, static inspection can bound delays to any number of cycles up to the execution latency of the mini-graph. One could construct a selector that accepts only mini-graphs with a potential serialization delay of 1 cycle (or less). In our experiments, the maximum execution latency of any mini-graph is 6 cycles, and *Struct-Bounded* accepts all mini-graphs with bounded serialization.

*Struct-Bounded* accepts strictly more mini-graphs than *Struct-None* and strictly fewer than *Struct-All*. Being only a heuristic, it rejects some mini-graphs with statically unbounded delay which are benign in practice; a delay may be statically unbounded but short or non-existent at run-time. It also accepts some harmful mini-graphs with bounded delay; even bounded delay is bad if it delays the resolution of a mispredicted branch.

### 4.3. Slack-Profile

Purely structural selectors—even heuristically serialization-aware ones—cannot determine whether serialization actually takes place and whether its delay degrades performance. Not all mini-graphs with the potential for serialization are actually delayed by the serializing input. And even if delay is induced, it might be masked by other delays and have no performance effect. *Slack-Profile* uses local slack profiles to quantify these structural unknowns.

*Slack-Profile* begins with a singleton execution schedule which details the ready times of all values and the issue times of all instructions. It then applies four simple rules to: (i) calculate the delay induced on an instruction by virtue of being placed in a mini-graph, and (ii) estimate whether a delay on a mini-graph's output can be absorbed by the program. Figure 5 uses an example to illustrate these rules. This section also discusses *Slack-Profile*'s profiling support and the rationale for its use of local, rather than global, slack.

**Quantifying mini-graph induced serialization delay.** As a singleton, an instruction's issue time is limited by the ready times of its inputs. As the first instruction in a mini-graph, an instruction waits for all inputs to the mini-graph. As a second (or later) instruction in a mini-graph, an instruction's issue time is determined by the issue time of the previous mini-graph instructions. For each mini-graph candidate, *Slack-Profile* uses rule #1 "external serialization" to determine the issue time of the first instruction and rule #2 "internal serialization" to determine the issue times of the remaining mini-graph instructions. For each instruction in the mini-graph candidate, the delay induced by mini-graph formation is the difference between the issue time of that instruction as a singleton and its issue time as part of the mini-graph. *Slack-Profile* uses rule #3 "instruction delay" to calculate this.

The bottom of figure 5 steps through a delay calculation.

**1: External Serialization**

$$Issue_{MG}(0) = \max_{i \in mg\text{-}inputs}(Ready(i), Issue(0))$$

**2: Internal Serialization**

$$Issue_{MG}(n) = Issue_{MG}(n-1) + Ex\text{-}Lat(n-1)$$

**3: Instruction Delay**

$$Delay_{MG}(n) = Issue_{MG}(n) - Issue(n)$$

**4: Performance Degradation**

$$Degrade_{MG} = \bigvee_{i \in mg\text{-}outputs}(Delay_{MG}(i) > Slack(i))$$

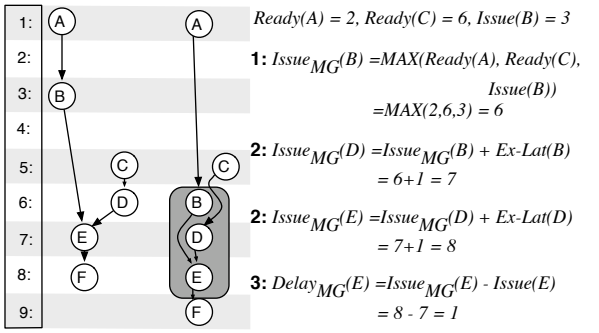



**FIGURE 5. Slack-Profile. Top:** delay and performance calculation rules. **Bottom:** An example calculation of the delay on instruction E induced by the formation of mini-graph BDE.

*Slack-Profile* starts with the singleton execution schedule on the left and calculates the mini-graph execution schedule on the right. The quantity of interest is the delay induced on instruction E by the formation of mini-graph BDE. The mini-graph has two external inputs, from instructions A and C, which are ready in cycles 2 and 6. Profile-Slack uses rule #1 to calculate the new issue time of instruction B, rule #2 to calculate the new issue times of instructions D and E, and rule #3 to calculate E's delay. The calculations agree with the depicted mini-graph schedule.

**Quantifying performance impact of delay.** Instruction delay degrades performance only if it cannot be "absorbed" by consuming instructions. The ability to absorb delay is formalized by *local slack* [7]. An instruction's local slack is the number of cycles by which it can be delayed without delaying any consumer. In Figure 5, instruction B has 3 cycles of local slack; it could be delayed 3 cycles without delaying E.

Slack-Profile uses per-static instruction local slack estimates to calculate whether a given mini-graph, if formed, will degrade performance. Specifically, a mini-graph will degrade performance if for any of its outputs—the profiler provides slack information for stores and branches and so these can be explicitly considered—the induced delay is greater than that output's local slack. This is rule #4. In the example in Figure 5, the formation of mini-graph BDE delays E by 1 cycle. BDE is rejected because E has a local slack of 0 cycles, and its delay is propagated to F.

**Profiling support.** As explained, *Slack-Profile* requires local slack estimates, issue times, and ready times. The slack profiling tool—in our case a simulator—generates these. It is important to note that issue and ready times are generated and used in the course of slack profiling. Acquiring these doesn't require heavier profiling, just more verbose profiler output[1]. The profiler outputs this information on a per-static instruction

basis, using averages over all profiled dynamic instances. Average issue and ready times for an instruction are reported relative to the issue time of the first instruction in its basic block (a convenient fixed reference point).

**Think globally, act locally.** In the strictest terms, delay on a given instruction translates into performance loss only if it consumes more than that instruction's *global slack*. However, our experience shows that local slack is a more useful indicator of mini-graph performance impact than global slack. The problem with global slack is that it relates all slack to a single critical path which can change with the introduction of a single mini-graph (a mini-graph's effect on fetch and commit bandwidth alone are enough to change the critical path through a program section). Global slack is more accurate than local slack on a per mini-graph basis. However, local slack is much less sensitive to the introduction of other mini-graphs. To effectively use global slack to select multiple mini-graphs requires re-profiling the program after every mini-graph is selected to obtain the new critical path and new global slacks. In contrast, effectively using local slack requires a single profile of a singleton (*i.e.*, non-mini-graph) execution.

### 4.4. Slack-Dynamic

*Slack-Dynamic* implements *Slack-Profile* in hardware. Rather than pruning the mini-graph candidate pool before selection, *Slack-Dynamic* identifies mini-graphs with harmful serialization at run-time and disables them. A mini-graph is disabled by restoring the original outlining jump in the instruction cache. *Slack-Dynamic* requires no profiling support and can detect serialization delays based on actual program behavior rather than qualitative heuristics or predictive models based on profiles of non-mini-graph runs.

Like *Slack-Profile*, *Slack-Dynamic* targets serialization directly as the root of performance loss; it considers a mini-graph harmful if it experiences serialization delay *and* if this serialization delays the execution of a consumer instruction.

To recognize when a mini-graph is actually delayed by a serializing input, *Dynamic-Slack* tracks last-arriving operands to mini-graphs. If a last arriving operand is a serializing operand (*i.e.*, an input to an instruction that isn't the first in the mini-graph) *and* if the mini-graph issues as soon as the operand arrives, serialization delay is flagged. Note, a simpler form of this serialization detection logic, one that only tracks operand arrival order and does not consider actual issue time relative to the arrival of the last operand, was first introduced as part of the dynamic "shotgun" critical path profiler [8] and later adopted as the serialization-avoidance heuristic by macro-op scheduling [15].

To determine whether a mini-graph's serialization delays a consuming instruction, *Slack-Dynamic* also tracks last-arriving operands and relative issue times for instructions that consume mini-graph output values. If a mini-graph consumer is delayed by a mini-graph which is itself serialized, the mini-graph is disabled.

Because execution schedules can change over dynamic instances, *Slack-Dynamic* uses a simple saturating counter hysteresis scheme to both avoid rashly disabling a mini-graph that serializes once and to support mini-graph resurrection.

**Downsides of dynamic pruning.** There are two potential downsides to dynamic pruning, at least in the context of mini-graphs. The first is reduced coverage. Static selection algorithms essentially restrict the pool of initial mini-graph candidates; instructions that reside inside in rejected candidates can still contribute to coverage as parts of other overlapping mini-graphs. In contrast, when *Slack-Dynamic* rejects a mini-graph, the lost coverage cannot be reclaimed because the singletons cannot be dynamically re-constituted into smaller mini-graphs. This downside is common to techniques that perform aggregation statically.

The second downside is reduced performance and is a function of mini-graphs' use of the "outlining" encoding scheme, which optimizes for mini-graph-enabled execution at the expense of mini-graph-disabled execution. Disabling an outlined mini-graph may remove execution serialization, but it introduces fetch serialization in the form of two additional jumps. In some cases, this exchange actually back-fires. An encoding scheme that supports high-performance dynamic pruning is a topic for future work.

## 5. Evaluation II: Serialization-Aware Selection

This section evaluates the proposed serialization-aware mini-graph selection algorithms *Struct-Bounded*, *Slack-Profile*, and *Slack-Dynamic*, and compares them to *Struct-All* and *Struct-None*. It also presents a detailed analysis of the algorithms using an exhaustive limit study and measures the robustness of slack profiles.

### 5.1. Performance and Coverage

The top graph in Figure 6 shows the performance of mini-graphs selected using the different schemes on the reduced processor. To provide additional insight, the middle and bottom graphs show relative performance on the fully-provisioned processor and coverage, respectively. The graphs include S-curves corresponding to the two naive selectors *Struct-All* and *Struct-None*.

**Struct-Bounded.** *Struct-Bounded* (diamond) behaves like a shifted version of *Struct-All* (black circle); its coverage and performance curves have the same basic shapes and slopes. *Struct-Bounded* provides less coverage and amplification than *Struct-All* (30% vs. 38%), but significantly better relative performance (–2% vs. –10%). Like *Struct-All*, *Struct-Bounded* has a performance "intersection" with *Struct-None*; however, that intersection is shifted to the left. By avoiding unbounded serialization, *Struct-Bounded* admits fewer pathological mini-graphs and the pathological mini-graphs it does admit do only "bounded harm". Whereas *Struct-All* induces slowdowns for 29 programs on the fully-provisioned processor, *Struct-Bounded* induces similar slowdowns on only 12 programs.

**Slack-Dynamic.** *Slack-Dynamic* (grey circle) has similar coverage (30%) to and slightly worse average performance (–6% on the reduced configuration and +6% on the fully-provisioned configuration) than *Struct-Bounded*. However, its performance and coverage curves have a shape and slope that resemble *Struct-None*, instead. *Slack-Dynamic* provides sub-

1. It is possible to reconstruct an execution schedule from local slack, but only for connected mini-graphs.

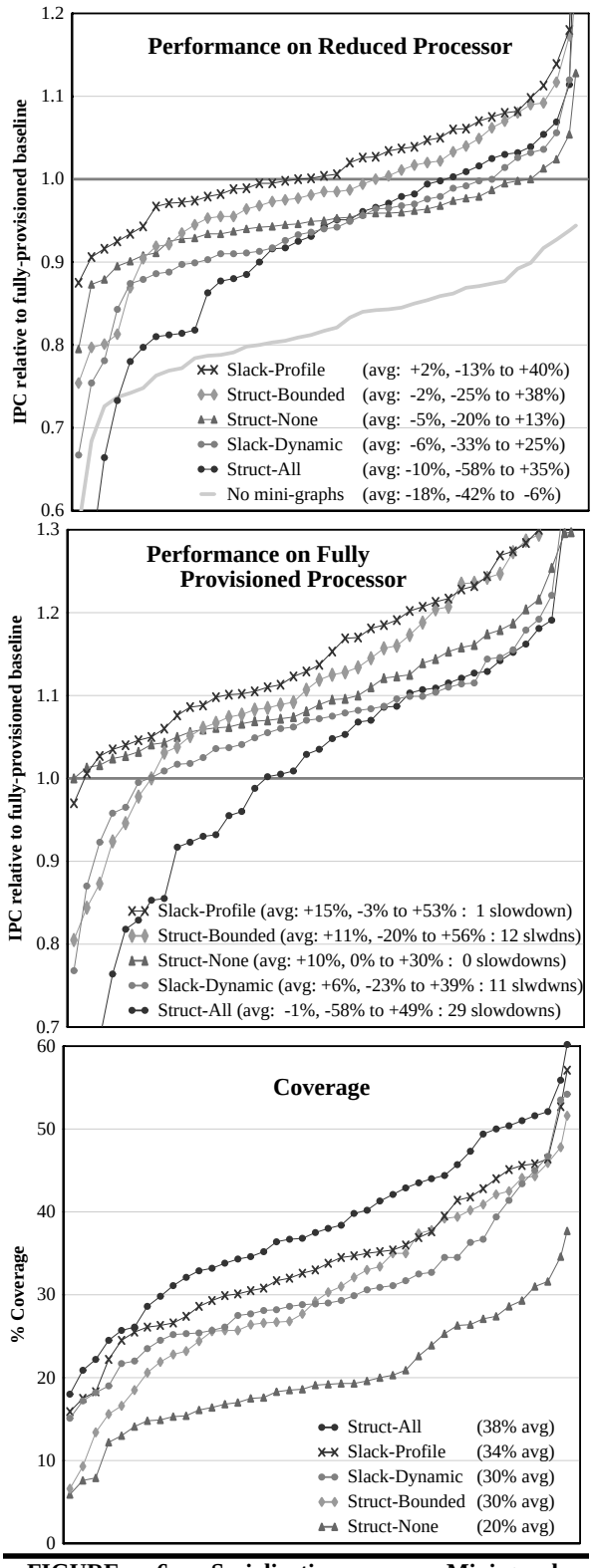



**FIGURE 6. Serialization-aware Mini-graph Selectors. Top:** Performance on reduced processor. **Middle:** Performance on fully-provisioned processor. **Bottom:** Coverage. *Slack-Profile* outperforms all other selectors and allows the reduced machine (with mini-graphs) to out-perform the fully-provisioned one (without mini-graphs).

stantially more coverage and slightly less performance.

*Slack-Dynamic* has lower coverage because it cannot reconstitute dynamically-disabled mini-graphs into smaller, more conservative mini-graphs. And although it out-performs *Struct-All*, *Slack-Dynamic* under-performs the other serialization-aware selection schemes. This is because although *Slack-Dynamic* eliminates execution serialization penalties, it effectively replaces them with fetch serialization penalties as a disabled mini-graph must execute in outlined form, which involves two jumps per instance.

**Slack-Profile.** *Slack-Profile* provides performance that is strictly superior to any other selection scheme, both on the reduced processor and the fully-provisioned processor[1]. *Slack-Profile* provides a 15% average performance improvement for the fully-provisioned processor and is the only selector capable of fully compensating (on average) for the reduced configuration. The reduced processor with *Slack-Profile* mini-graphs outperforms the fully-provisioned baseline by 2%. And although 35% of programs experience some performance loss relative to the fully-provisioned processor, only 15% experience more than a 5% loss. For comparison, with no mini-graphs, every program experiences *at least* a 5% loss.

The key to *Slack-Profile*'s success is its combination of aggressive coverage (34% on average) and intelligent serialization avoidance. On the reduced configuration, it is aggressive enough to outperform *Struct-All* on the right side of the graph, and selective enough to outperform *Struct-None* on the left. On the fully-provisioned processor, where amplification benefits are lower and serialization costs are exposed, it provides the additional coverage needed to consistently out-perform *Struct-None. Slack-Profile* is the "target selector" we earlier argued should exist.

## 5.2. Slack-Profile: Breaking Down the Model

The *Slack-Profile* model has two components: mini-graph-induced instruction delay (rules #1–3) and impact of delay on consumer instructions (rule #4). The top graph in Figure 7 isolates the contribution of these components. For comparison, the graph also shows full *Slack-Profile*, *Struct-None*, and *Struct-All*.

**Contribution of "consumer delay".** *Slack-Profile-Delay* corresponds to a partial model that does not include rule #4. This model rejects mini-graphs whose output is delayed, regardless of whether that delay can be absorbed by consumer instructions. It generates a strictly smaller mini-graph candidate pool than *Slack-Profile*. On average, explicit accounting for the impact of delay on consumers contributes 1% performance on the reduced configuration. However, it does change the slope of *Slack-Profile*, contributing 4% for some programs on the right side of the graph while actually reducing the performance of a few programs on the left. Here, slightly reduced coverage is exposing the few pathological mini-graphs that both *Slack-Profile* and *Slack-Profile-Delay* admit.

**Contribution of "serialization delay".** The difference

1. The lone exception is *mcf* on the fully-provisioned machine, where *Slack-Profile* is outperformed by *Struct-None*. *Slack-Profile* uses optimistic execution latencies that do not account for cache misses, which plague *mcf*. Remedying this is left for future work.

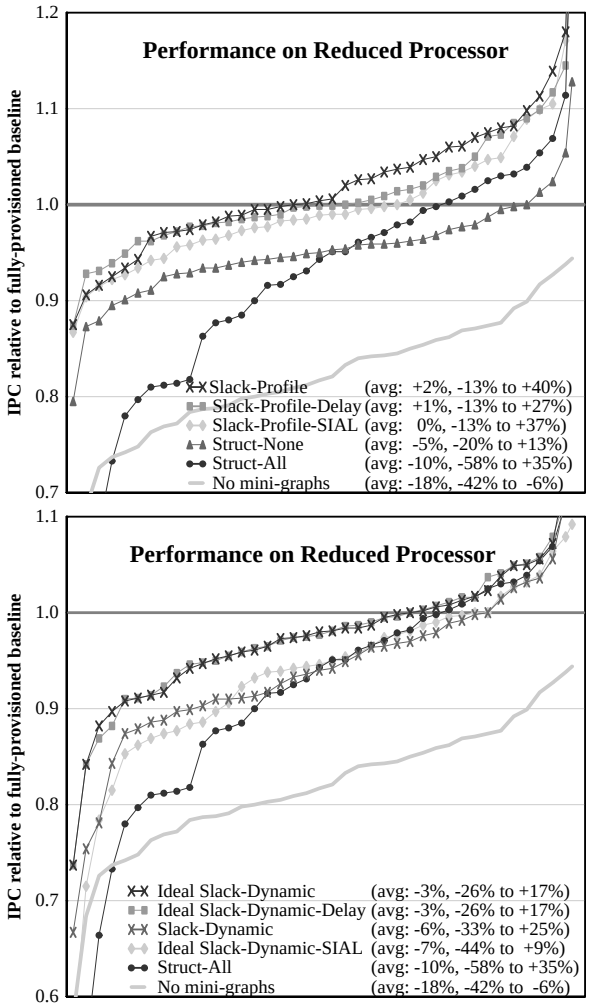



**FIGURE 7. Isolating Components of the Models. Top:** *Slack-Profile*. **Bottom:** *Slack-Dynamic*.

between *Slack-Profile-Delay* and either *Struct-All* or *Struct-None* corresponds to the contribution of explicit accounting of serialization delay. Obviously, it is this component of the model that accounts for bulk of *Slack-Profile*'s advantage over serialization-blind selection schemes.

*Slack-Profile-SIAL* (SIAL=Serial Input Arrives Last) is a variant of *Slack-Profile-Delay* that ignores actual issue delays and focuses only on operand arrival times. Some aggregation schemes [15] use SIAL as their serialization-avoidance heuristic. The difference between *Slack-Profile-Delay* and *Slack-Profile-SIAL* (4% on average) shows that explicit accounting for delay is preferred to the operand-arrival-order heuristic.

### 5.3. Dynamic Slack: Breaking Down the Model

Like *Slack-Profile*, the *Slack-Dynamic* serialization avoidance model also has two components: serialization delay and impact on consumers. *Slack-Dynamic* also has additional performance components that correspond to the outlining penalty for disabled mini-graphs and loss of coverage. Loss of coverage is difficult to isolate, but the bottom graph of Figure 7 attempts to isolate the other three.

**Contribution of "outlining" penalty.** *Ideal-Slack-Dynamic* is an implementation of *Slack-Dynamic* in which the "outlining" penalty for disabled mini-graphs is removed. This curve isolates the performance of *Slack-Dynamic*'s model from the performance effects of the mini-graph encoding scheme. On average, the performance penalty of "outlining" degrades *Slack-Dynamic*'s performance by 3% on the reduced processor. Without this penalty, *Slack-Dynamic* is much more competitive with *Slack-Profile*.

**Contribution of "consumer impact".** *Ideal-Slack-Dynamic-Delay* is a penalty-free *Slack-Dynamic* selector that considers only serialization delay, not impact on consuming instructions. This model disables more mini-graphs than *Slack-Dynamic*. The contribution of explicit accounting for the potential absorption of delay is the difference between this selector and *Ideal-Slack-Dynamic*. Interestingly, in the idealized *Slack-Dynamic* case, explicit consideration of consumer impact makes less than 1% difference on average. However, when *Slack-Dynamic* is used with outlining penalties, this component of the model contributes almost 2% to performance. Here, explicit accounting avoids disabling mini-graphs over-aggressively and incurring their outlined execution penalty.

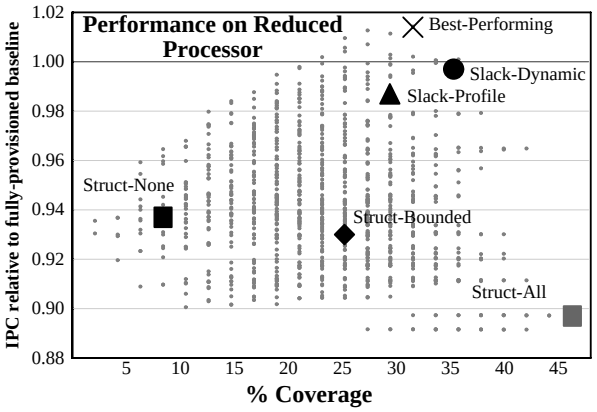
**Contribution of "serialization delay".** The contribution of *Slack-Dynamic*'s serialization delay detection component is the difference between *Ideal-Slack-Dynamic-Delay* and *Struct-All*. As in *Slack-Profile*, delay accounting provides the bulk of *Slack-Dynamic*'s impact. *Ideal-Slack-Dynamic-SIAL* again compares true delay accounting with heuristic tracking of relative operand arrival times. Again, explicit delay accounting provides superior performance.

### 5.4. Analysis: Comparison with Exhaustive Search

Mini-graph selection algorithms choose at most 512 mini-graph templates from tens of thousands of candidates. Because of the huge number of possible combinations and because the choice of one template affects the pool of remaining templates, it is not computationally feasible to perform a traditional "limit study" to determine the ideal set of mini-graphs for each program. To analyze our selection algorithms, we create a significantly smaller search space over which we can exhaustively search. For one short-running benchmark (*adpcm.c*) we choose the 10 most frequently occurring non-overlapping static mini-graphs. We then evaluate all 1024 combinations of mini-graphs and compare the sets chosen by each selector to the best performing set chosen by exhaustive search.

The graph of Figure 8 shows a coverage (x-axis) and performance (y-axis) scatter plot for the 1024 possible mini-graph sets running on the reduced configuration. Again, because only 10 static mini-graphs are considered, this data does not represent the actual coverage or performance for *adpcm.c*. The results of each selection algorithm is highlighted at its coverage/performance intersection.

The results are strikingly intuitive. *Struct-All* (light box) occupies the right-most point in the graph; it includes all 10 mini-graphs. *Struct-None* (dark square) occupies the left-most, *i.e.*, lowest coverage, point among all non-exhaustive selectors. *Struct-Bounded* (diamond), which heuristically allows bounded serialization, yields decent coverage but poor perfor-




| ADPCM mgs: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Struct-None | Y | Y | | | | | | | | |
| Struct-Bounded | Y | Y | Y | Y | Y | | | | | |
| Slack-Profile | Y | Y | Y | | | Y | Y | Y | Y | |
| Struct-All | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y |
| *Best Performing* | | Y | | Y | | Y | Y | | Y | |
| good in isolation? | Y | Y | Y | Y | | Y | Y | Y | Y | |

**FIGURE 8. Limit study and analysis. Top:** coverage vs. performance scatter plot of all possible combinations of 10 mini-graph candidates for *adpcm.c*. Mini-graph combinations chosen by the five selectors are highlighted. **Bottom:** Detailed selector choices.

mance. The quantitative slack-based selectors, *Slack-Profile* (triangle) and *Slack-Dynamic*, combine high coverage with high performance, with both approaching the performance of the ideal mini-graph set obtained by exhaustive search.

One unintuitive result is the position of *Slack-Dynamic*, especially relative to *Slack-Profile*. In a realistic selection scenario, *Slack-Dynamic* has poorer coverage and performance than *Slack-Profile* because of its inability to re-constitute dynamically disabled mini-graphs into smaller, benign alternatives. In this less realistic experiment, this disadvantage is eliminated because the initial pool consists only of non-overlapping mini-graph candidates.

**Examining selector choices.** The table in Figure 8 shows each selector's set of chosen mini-graphs. None of our selectors picks the best performing set (X in the graph). The first two mini-graphs (0,1) are non-serializing and are automatically included by all selectors. *Struct-Bounded* admits three mini-graphs whose serialization is bounded, for a total of five (0-4). *Slack-Profile* rejects three mini-graphs whose estimated delay is larger than the output slack, including two mini-graphs with bounded serialization that were accepted by *Struct-Bounded* (3,4). It selects a total of seven (0-2, 5-8).

The best performing set includes a mini-graph which *Slack-Profile* rejects (3). Although this mini-graph is delayed a full cycle more than its slack, it improves performance. This is because its constituent instructions are fetch critical over 50% of the time; coalescing them into a handle creates enough slack to compensate for the delay. *Slack-Profile* does not account for fetch criticality—none of our models do—and thus has no means of assessing such a trade-off.

*Slack-Profile* includes a mini-graph which the best performing set excludes (7). When chosen in isolation, this mini-graph actually improves performance! That mini-graph 7 is not included in the best performing set speaks to the fact that mini-graph selection is non-decomposable. (Similarly, 0, a mini-graph without any serialization vulnerability, is not included in the best performing set!) Estimating how a mini-graph might change the execution of an unmodified program is not always an accurate indicator as to how it might change the execution of a program with other mini-graphs. Because *Slack-Profile* does not re-profile local slack after each mini-graph is selected, it assesses mini-graphs in isolation without considering potential interference.

### 5.5. Robustness of Slack Profiles

In the experiments so far, *Slack-Profile*—the best performing selector—used "self-trained" profiles collected from simulations on the target configuration (the reduced processor) and on the target program data input set. Although self-training with respect to the target microarchitecture is realistic, self-training with respect to the input data set is not; inputs vary across dynamic program invocations. Here, we measure the robustness of slack profiles across microarchitecture configurations and program input data sets.

**Robustness to machine configuration.** Intuitively, the main determinants of performance (and subsequently of slack) are the dataflow graph, the latency of the memory system (via cache misses) and branch predictor (via mis-predictions), and the capacity and bandwidth of the pipeline. Of these, the factors that are most likely to vary across machines are pipeline bandwidth and capacity, and on-chip memory-system capacity.

The top graph in Figure 9 shows the performance S-curve for MediaBench and CommBench programs running self-trained *Slack-Profile* mini-graphs on the reduced processor. In addition to this S-curve, the graph also shows results for mini-graphs cross-trained on three configurations: (i) a further reduced 2-way issue processor (*cross 2-way*, circle), (ii) an 8-way issue processor (*cross 8-way*, triangle), and a reduced processor with an 8KB data cache and a 256KB L2 (*cross dmem/4*, diamond). These are not shown as S-curves, but rather as points at the same horizontal position as the corresponding program on the S-curve.

The performance of cross-trained mini-graphs is stable across these three configurations, suggesting that slack profiles are robust across a relatively wide range of realistic microarchitectures. This is evidenced by the fact that most points lie directly under the self-trained S-curve. Performance is occasionally somewhat lower—and coverage somewhat higher—for mini-graphs selected using the 2-way issue profile (circle). This is intuitive. The 2-way issue processor generally yields more execution slack as ready instructions wait due to limited issue bandwidth. A profile generated on a relatively reduced machine will result in the selection of a few harmful mini-graphs whose delay cannot be absorbed on a more provisioned processor that typically generates less slack.

**Robustness to input sets.** Intuitively, slack profiles should be insensitive to inputs. Slack is largely a product of parallelism, branch predictability, and memory behavior. These factors are regarded as being program—not input—specific.

The bottom graph of Figure 9 shows performance results for SPECint and MiBench programs running *Slack-Profile* mini-graphs on the reduced processor. In addition to the self-

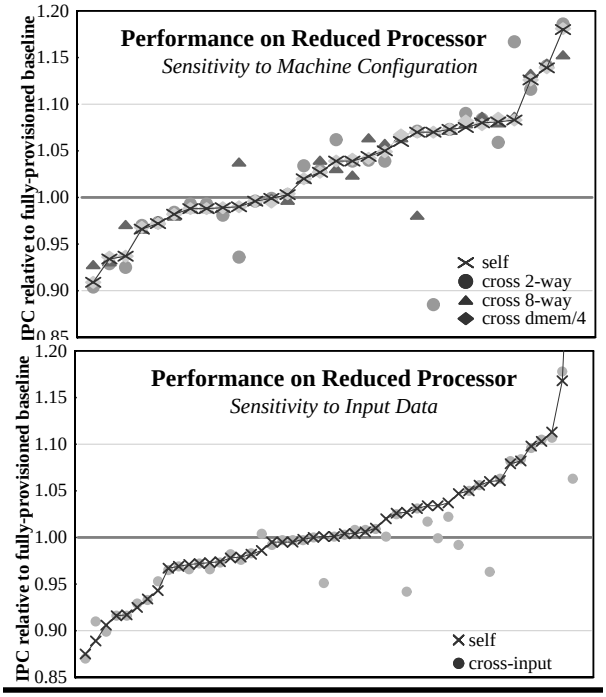



**FIGURE 9. Slack profile robustness. Top:** Micro-architecture sensitivity. Profiles generated with 2-, 8- way machines, as well as a machine with 1/4 the cache size. **Bottom:** Program input data set sensitivity.

trained mini-graphs S-curve (*self*), there are points for mini-graphs cross trained on a different data input set (*cross-input*, circle): *ref* for SPECint and *small* for MiBench. Again, there is little difference (less than 2% absolute, on average) in both coverage and performance between self-trained and cross-trained mini-graphs, suggesting that slack profiles are robust across program input data sets. Where variation does occur, the reason is differences in code coverage between the two runs. Such differences can arguably be eliminated by training on multiple input sets that exercise most of the static code.

### 5.6. Applicability to Other Aggregate Schemes

Although presented and evaluated in the context of mini-graphs, the selection algorithms presented here should be applicable to other aggregation schemes that target dynamically scheduled processors [4, 15, 18]. Different schemes tend to offer different benefits, depending on the underlying microarchitecture, the pipeline stages and structures that exploit aggregation, aggregate size and interface restrictions, and the use (or lack thereof) of custom aggregate acceleration. Aggregation costs, on the other hand, are common to many aggregation schemes. External serialization is almost unavoidable with the use of aggregation in a dynamically-scheduled context. Internal serialization is less fundamental—although several aggregation schemes do suffer from it—but it is also a lesser effect.

Our selection algorithms should be applicable to other aggregation schemes because they focus on the aspect of aggregation that is common to all of them: the cost of serialization. And although as presented, *Struct-Bounded* and *Slack-Profile* explicitly model internal serialization, this aspect of the model can be easily removed if it does not apply. *Slack-Dynamic* only implicitly accounts for internal serialization.

Our algorithms do not explicitly account for mini-graph performance benefit; partly because the indirect benefits of resource amplification are smaller than the direct costs of serialization and partly because they are also more difficult to model. However, the models can be easily extended to account for direct performance benefits, like ones provided by custom latency reduction [4, 18] or explicit tolerance of some latency in the underlying micro-architecture, *e.g.*, a pipelined scheduler [15].

Our early experiments with macro-op scheduling, indicate that *Slack-Dynamic* (modulo the penalties of outlining and reduced coverage) is a more selective, better performing serialization-filter than macro-op's own heuristic (SIAL, Serial Input Arrives Last) and that a retargeted version of *Slack-Profile* produces better results than the heuristics used by macro-op execution [11]. The SIAL results in Sections 5.2 and 5.3 provide additional support for this observation, albeit in the context of mini-graphs, not macro-ops.

## 6. Related Work

Mini-graphs are instruction aggregates [5, 18, 20] that target dynamically-scheduled superscalar processors. Many proposed aggregation techniques use custom functional units to reduce the latency of graphs of arithmetic operations [10, 22, 23]. These generally target simple in-order pipelines.

Mini-graph processing is not alone in exploiting aggregation for superscalar resource amplification, but it is more general than other techniques. AMD's Athlon [6] breaks some integer operations apart at the issue stage, bypassing the need to rename and issue RISC micro-instructions individually. Intel's Pentium M [13] fuses load/execute and store-address/-data micro-op pairs, reducing the number of x86 instructions that decode into multiple micro-op sequences, and the number of micro-ops renamed, scheduled, and committed, and amplifying issue queue capacity. Extensions to the x86 ISA for fusing instruction pairs have also been proposed [11, 12]. Macro-op scheduling [15] micro-architecturally fuses instructions in order to boost scheduling capacity and hide scheduling loop latency, but does not amplify the bandwidths or capacities of any other structures. Dynamic strands [18] extend macro-op scheduling beyond pair-wise fusion and execute on closed-loop ALUs, yielding a transparent, partial Instruction Level Distributed Processing (ILDP)[14] implementation. Static strands [19] are similar to mini-graphs, but do not amplify instruction cache capacity or pipeline bandwidth, because strand instructions are aggregated after fetch.

A few aggregation proposals address the subject of serialization. Macro-op scheduling [15] avoids "harmful grouping" by disabling macro-ops whose serializing input is the last-arriving operand. *Slack-Dynamic* detects actual issue delay and the impact it has on dependent instructions. Macro-op execution [11] uses a two-pass fusing algorithm to dynamically fuse x86 micro-ops. This algorithm is based on heuristics for instructions that tend to be on the critical path (*e.g.*, ALU operations), for ones that do not (*e.g.*, stores), and for instruction pairs that are likely to be critical (*e.g.*, instructions near each other in the original x86 code). Like *Struct-Bounded*, two-pass

fusing is heuristic and does not actually determine the critical path of the program nor does it quantify the potential for or cost of serialization. Configurable Compute Accelerator (CCA) graphs [4] target performance improvement via custom or pseudo-custom acceleration. CCA graph selection uses "slack" as a dataflow graph traversal tie-breaker, not as a performance diagnostic. CCA exploits graph latency reduction which tends to mask serialization problems.

## 7. Conclusions

Instruction aggregation is a technique that can be exploited to amplify the capacity and bandwidth of dynamically scheduled superscalar processors, theoretically improving the IPC throughput that can be achieved with a given amount of physical resources. Mini-graphs are a particular form of instruction aggregation that targets resource amplification throughout the entire pipeline.

Aggregation techniques are afflicted by serialization, which delays the issue and execution of instructions within aggregates. The most pervasive and destructive form of serialization is external serialization, which forces an aggregate to wait for all of its external inputs before it can issue. Serialization can degrade performance if not explicitly accounted for during the aggregate formation process.

This paper examines serialization in the context of mini-graph processing—where serialization's negative effects are especially pronounced—and develops three mini-graph selectors that identify and reject mini-graphs with performance-degrading serialization. The most promising of these is *Slack-Profile*, which uses local slack profiles to calculate mini-graph induced delay and reject only mini-graphs whose delay exceeds the program's ability to absorb it. *Slack-Profile* achieves coverage and amplification rates of 34% while virtually eliminating serialization penalties. This combination allows a 3-way issue processor with *Slack-Profile* mini-graphs to (on average) outperform a 4-way issue processor by 2%.

## Acknowledgments

The authors thank the anonymous reviewers for their comments. This work is supported by NSF CAREER award CCF-0238203, and by grants from the Intel Research Council.

## References

[1] A. Bracy, P. Prahlad, and A. Roth. "Dataflow Mini-Graphs: Amplifying Superscalar Capacity and Bandwidth." In *Proc. 37th International Symposium on Microarchitecture*, pages 18–29, Dec. 2004.

[2] A. Bracy and A. Roth. "Outlining: A Compatible, High-Performance Mini-Graph Encoding Scheme." Technical Report TR-CIS-06-11, University of Pennsylvania, Aug. 2006.

[3] J. Butts and G. Sohi. "Characterizing and Predicting Value Degree of Use." In *Proc. 35th International Symposium on Microarchitecture*, Nov. 2002.

[4] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner. "Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization." In *Proc. 37th International Symposium on Microarchitecture*, pages 30–40, Dec. 2004.

[5] N. Clark, H. Zhong, and S. Mahlke. "Processor Acceleration through Automated Instruction Set Customization." In *Proc. 36th International Symposium on Microarchitecture*, pages 129–140, Dec. 2003.

[6] K. Diefendorf. "K7 Challenges Intel." *Microprocessor Report*, 12(14), Nov. 1998.

[7] B. Fields, R. Bodik, and M. Hill. "Slack: Maximizing Performance under Technological Constraints." In *Proc. 29th International Symposium on Computer Architecture*, pages 47–58, May 2002.

[8] B. Fields, S. Rubin, and R. Bodik. "Focusing Processor Policies via Critical Path Prediction." In *Proc. 27th Annual International Symposium on Computer Architecture*, pages 74–85, Jul. 2001.

[9] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. "MiBench: A Free, Commercially Representative Embedded Benchmark Suite." In *4th Workshop on Workload Characterization*, Dec. 2001.

[10] J. Hauser and J. Wawrzynek. "Garp: A MIPS Processor with a Reconfigurable Coprocessor." In *Proc. 1997 IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 1997.

[11] S. Hu, I. Kim, M. Lipasti, and J. Smith. "An Approach for Implementing Efficient Superscalar CISC Processors." In *Proc. 12th International Symposium on High-Performance Computer Architecture*, pages 41–52, Jan. 2006.

[12] S. Hu and J. Smith. "Using Dynamic Binary Translation to Fuse Dependent Instructions." In *Proc. 2nd International Symposium on Code Generation and Optimization*, Mar. 2004.

[13] Intel Corporation. *Mobile Intel Pentium 4 M-Processor Datasheet*, Jun. 2003. http://www.intel.com/design/mobile/datashts/250686.htm.

[14] H.-S. Kim and J. Smith. "An Instruction Set and Microarchitecture for Instruction Level Distributed Processing." In *Proc. 29th International Symposium on Computer Architecture*, May 2002.

[15] I. Kim and M. Lipasti. "Macro-op Scheduling: Relaxing Scheduling Loop Constraints." In *Proc. 36th International Symposium on Microarchitecture*, pages 277–288, Dec. 2003.

[16] C. Lee, M. Potkonjak, and W. Mangione-Smith. "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems." In *Proc. 30th International Symposium on Microarchitecture*, pages 330–335, Dec. 1997.

[17] S. Liao, S. Devadas, and K. Keutzer. "A Text-Compression-Based Method for Code Size Minimization in Embedded Systems." *ACM Transactions on Design Automation of Electronic Systems*, 4(1):12–38, 1999.

[18] P. Sassone and D. Wills. "Dynamic Strands: Collapsing Speculative Dependence Chains for Reducing Pipeline Communication." In *Proc. 37th International Symposium on Microarchitecture*, pages 7–17, Dec. 2004.

[19] P. Sassone, D. Wills, and G. Loh. "Static Strands: Safely Exposing Dependence Chains for Increasing Embedded Power Efficiency." In *Proc. 2005 Conference on Languages, Compilers, and Tools for Embedded Systems*, Jun. 2005.

[20] F. Spadini, M. Fertig, and S. Patel. "Characterization of Repeating Dynamic Code Fragments." Technical report, University of Illinois, Center for Reliable and High-Performance Computing, 2002.

[21] T. Wolf and M. Franklin. "CommBench: A Telecommunications Benchmark for Network Processors." Technical Report WUCS-99-29, University of Washington in St. Louis, Nov. 1999.

[22] Z. Ye, A. Moshovos, S. Hauck, and P. Banerjee. "CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit." In *Proc. 27th International Symposium on Computer Architecture*, Jun. 2000.

[23] S. Yehia and O. Temam. "From Sequences of Dependent Instructions to Functions: A Complexity-Effective Approach for Improving Performance without ILP or Speculation." In *Proc. 31st International Symposium on Computer Architecture*, pages 159–170, Jun. 2004.