MINI-GRAPH PROCESSING

Anne Weinberger Bracy

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2008

Amir Roth
Supervisor of Dissertation

Rajeev Alur
Graduate Group Chairperson

Acknowledgements

"It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of despair, we had everything before us, we had nothing before us...." – Charles Dickens

Enough of the worst! Let me reflect only on the best. My graduate career led me to the best of all possible friends, professors, and colleagues. My mind is sharper, my life richer, and my heart fuller thanks to the years I spent at Penn, learning, living, and loving more than any stretch of time prior to graduate school.

I thank Amir Roth, whose passion for teaching and computer architecture converted me from a forced pupil in a required class to an inspired, budding researcher with a thirst for more knowledge and an empowering feeling that anything is possible in hardware. You have been a father and a friend. Thank you for setting the bar high and then helping me reach it.

I thank my committee chair, Milo Martin, for always having an open door and wise council. I also thank the members of my committee (past and present)—E Christopher Lewis, David Albonesi, Andre DeHon, and Insup Lee—for your expertise, advice, and encouragement.

My academic siblings—Vlad Petric, Tingting Sha, and Drew Hilton—have been the truest of friends. They are the ying to my yang, my one and only *Mei Mei*, and my knight in shining armor, respectively. My office mates—Marc Corliss, Colin

Blundell, Prashant Prahlad, Joe Devietti, and Arun Raghavan—have been a source of strength and hilarity throughout my years at Penn. I will never forget the many late nights, early mornings, happy hours, and the wide variety of humor we shared in the most colorful, chaotic, and decked out office in Levine. J. Adam Butts generously made his degree of use framework available to me, for which I am very grateful.

I met many dear friends in graduate school: Ryan McDonald, Ameesh Makadia, Andrew McGregor, Tania Patsialou, Nick Montfort, Boulos Harb, Sid Suri, John Blitzer, Margaret DeLap, Aaron Evans, Katie Candland, and Jennifer Plummer. Christie Avraamides was hands down the best roommate I could ever have hoped for in my graduate school experience. Thanks for always reminding me to put jobs on the cluster before we went out at night. Patrick Coffman, Matt Ginzton, and Gabriel Adauto kept the Bay Area home fires burning for me. Erica Robles, Ethan Schuchman, and Hanna Wallach saved me hundreds of hours and thousands of dollars in therapy.

I am grateful to many people at Penn. Mike Felker has been a source of care and an administrative lifesaver from day one. Rita Powell nurtured the women in the department in countless ways and was always my personal cheerleader. Benjamin Pierce is a friend whose art and friendship I appreciated both in Levine and throughout Philadelphia. Dan Widyono kept our clusters up and running.

I thank my colleagues at the Microarchitectural Research Lab at Intel (past and present)—Hong Wang, Quinn Jacobson, John Shen, Drew Alduino, Joe Schutz, and Shekhar Borkar—for showing me the light at the end of the tunnel. Thanks also for the great confidence you had in me so early in my career.

Thanks to my family—Mom, Dad, Layne, Amy, Seth, Becca, Kevin, Andrew, Akiko, and Andy—for reminding me of the things that truly matter most. Mom and Dad, I could not have made it this far without your unconditional love and support. Thanks to the Weinberger clan—Anna Maria, Sabine, Lorenz, Fabian, Katharina, Bastian, and Suzanne—for encouraging me all these years. I am honored to have

become a Weinberger in the middle of my graduate career.

A special thanks to my little Timo Quilliam Weinberger (my Mini-Graf?) for putting up with my dissertating for your entire thus-far mini-existence.

Finally, and most importantly, I thank my companion, husband, and best friend, Kilian Quirin Weinberger. You cheered me on; you cheered me up. You believed in me. In you I have found love, peace, laughter, and excitement for what lies ahead.

ABSTRACT

MINI-GRAPH PROCESSING

Anne Weinberger Bracy

Amir Roth

For years, single-thread performance was the most dominant force driving processor development. In recent years, however, the poor scaling of single-thread superscalar performance and power concerns coupled with the ever-increasing number of transistors available on chip has changed the focus from single-thread performance to thread-level parallelism running on multi-core designs. The trend is for these cores to be narrower with smaller windows. This dissertation addresses the question of how to maintain—and, ideally, improve—single-thread performance under such constraints.

Mini-graph processing is a form of instruction fusion — the grouping of multiple operations into a single processing unit — that increases the instruction-per-cycle (IPC) throughput of dynamically scheduled superscalar processors in an efficient way. *Mini-graphs* are compiler-identified aggregates of multiple instructions that look and behave like singleton instructions at every pipeline stage, except for execute — there the constituent operations are retrieved and performed serially micro-code style. A mini-graph processor exploits instruction fusion to increase the efficiency of pipeline stages and structures that perform instruction book-keeping.

This dissertation describes a mini-graph architecture and evaluates it using cycle-level simulation. A superscalar processor enhanced with mini-graphs can match the performance otherwise achieved with a wider, deeper superscalar processor. Experiments show that across four benchmark suites, the addition of mini-graph processing allows a dynamically scheduled 3-wide superscalar processor to match the IPC of a 4-wide superscalar machine.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Mini-graph processing is a form of instruction fusion — the grouping of multiple operations into a single processing unit — that increases the instruction-per-cycle (IPC) throughput of dynamically scheduled superscalar processors in an efficient way [12, 14]. A *mini-graph* is an atomic multi-instruction aggregate with the interface of a RISC singleton instruction that is processed as a single instruction at every pipeline stage, except for execute — there the constituent operations are retrieved and performed serially micro-code style. A software tool statically identifies mini-graphs and encodes them in the program binary on a per-application basis. Mini-graph processing improves IPC by amplifying the capacities and bandwidths of structures and stages that perform instruction bookkeeping and inter-operation register communication.

This dissertation describes a mini-graph architecture and evaluates it using cycle-level simulation. Experiments show that across four benchmark suites, the addition of mini-graph processing allows a dynamically scheduled 3-wide superscalar processor to match the IPC of a 4-wide superscalar machine.

1.1 Efficient Uniprocessors

For years, single-thread performance was the most dominant force driving processor development. Performance was pursued through a combination of increasing clock frequencies and higher IPC throughput via wider, superscalar issue and more deeply speculative out-of-order execution.

In recent years, however, this approach has curtailed for two reasons. First, concerns about power have slowed the push for ever increasing frequencies. Second, IPC improving techniques such as pipeline and instruction window scaling have diminishing returns on investment. Not only is the marginal benefit of an additional issue slot, for example, sub-linear in terms of IPC, but it also diminishes; the n^{th} slot is less utilized than the $n-1^{th}$. At the same time, the cost in terms of area, power, and delay is super-linear; the n^{th} slot costs more than the $n-1^{th}$ slot, because it raises the cost of the existing slots by requiring communication and coordination with them. The marginal benefit of an additional window entry is similarly sub-linear. As a consequence, many aspects of processor design are becoming less aggressive. Pipelines hit maximum depth with the Pentium 4 and subsequent designs have fewer pipeline stages. Some superscalar designs have been scaled to 5- and 6-wide [37, 46, 50, 93], but 4-wide appears to be the superscalar sweet-spot [106].

In response to the poor scaling of single-thread superscalar performance and the ever-increasing number of transistors available on chip, industry has decreed that future performance scaling will come from thread-level parallelism running on multi-core designs. This approach places a premium on the area and power efficiency of individual cores. The trend, therefore, is towards narrower cores with smaller windows; but doing so lowers single-thread performance, a side-effect industry cannot afford. The question then remains, how to maintain—and, ideally, improve—single-thread performance under such constraints.

1.2 Instruction Fusion

Single thread performance is measured as the execution time of particular program:

$$\text{program latency} = \frac{\text{dynamic instruction count}}{IPC \times \text{clock frequency}}$$

where IPC is the average number of instructions processed per cycle, and clock frequency is measured in cycles per second. Traditional approaches to performance improvement either increase clock frequency or throughput (IPC). This dissertation leverages instruction fusion, an alternative approach to performance improvement that focuses instead on the third component of the above equation: dynamic instruction count.

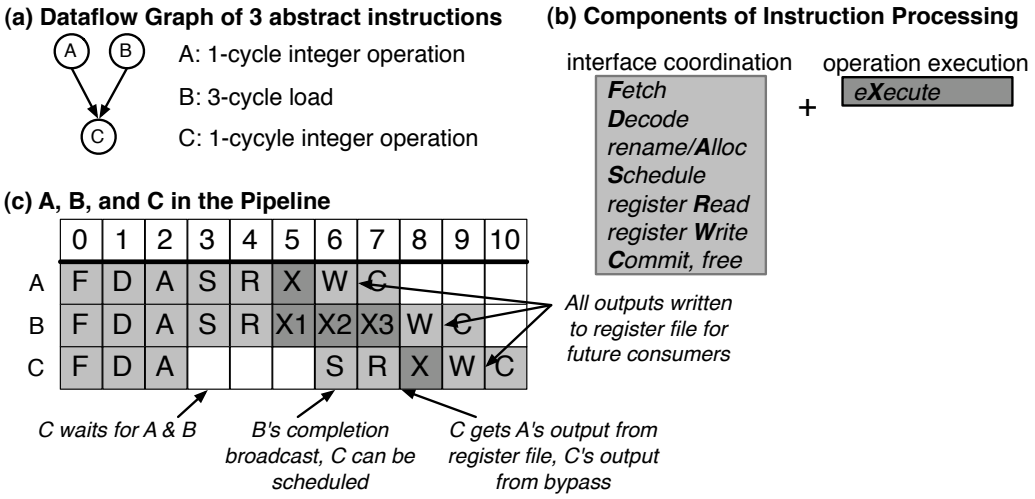


Figure 1.1: Instruction Processing: Interface Coordination + Operation Execution

Instruction = Interface + Operation. To reduce the dynamic instruction count, instruction fusion exploits the distinction between two aspects of instruction processing: operation execution and interface coordination. Consider the instruction "`add r1, r2` → `r3`." Semantically, this instruction requires that an addition operation takes place. But processing this instruction is not just a matter of addition; it also concerns the instruction's inputs and outputs. Figure 1.1a shows the dataflow

graph of three abstract instructions, `A`, `B`, and `C`. Figure 1.1b lists the pipeline stages required to process these instructions, separating the interface coordination (left) from the actual execution (right). Figure 1.1c shows these three instructions as they travel down the pipeline. At the execute stage (shaded), the instructions' actual operations are executed. Before and after executing the operations, a processor performs book-keeping and inter-operation communication steps. The operation is only a small component of the dynamic processing of an instruction; book-keeping and communication steps dominate with respect to cycles, pipeline stages, and on-chip resource usage.

Instruction fusion and resource amplification. The faithful execution of a program requires that the semantics specified by the operations in a binary not be changed. Interface coordination, however, is left to the processor's discretion. It is these coordinating steps that are combined in instruction fusion. Instruction fusion groups multiple operations into a single processing unit for select pipeline stages. During these stages, book-keeping and communication steps are performed on the aggregate rather than the individual operations.

Instruction fusion decreases the number of book-keeping and communication steps required per operation. The operation count is unaltered, but the number of interface-coordinating steps is decreased. Instruction fusion does not suffer from the diminishing performance returns on resource investment associated with wider-issue, larger-instruction window superscalar processors. Instruction fusion does not increase the physical complexity (latency, area, power) of on-chip structures; on the contrary, it amplifies resources, and allows superscalar structures to be made simpler. Reducing the number of book-keeping steps required to execute a program *amplifies* the capacities and bandwidths of those resources dedicated to instruction book-keeping. Bandwidth amplification allows younger instructions to enter a particular stage earlier. For example, amplifying issue bandwidth allows some instructions to potentially issue sooner. Capacity amplification of a given structure potentially

allows younger instructions to enter the structure sooner. For example, amplifying the register file allows instructions that might previously have stalled waiting for a free register to be renamed earlier.

The benefit of instruction fusion is proportional to its *dynamic coverage*, the percent of dynamic instructions embedded into aggregates. Coverage measures the resource amplification introduced by instruction fusion. A program with 30% dynamic coverage, for example, has approximately 30% fewer instructions at those pipeline stages for which fusion takes place. Coverage is defined by the following equation:

$$\text{dynamic coverage} = \sum_{\forall mini-graphs}, \frac{frequency(\text{mini-graph}) \times (size(\text{mini-graph}) - 1)}{\text{program's dynamic instruction count}}$$

where *size* is the length of the mini-graph in number of instructions, and $frequency$ is the number of times each mini-graph is encountered throughout the execution of a program. Each time a 4-instruction mini-graph is encountered, for example, one instruction is processed where once there were 4 instructions; three instructions are internal to the mini-graph. Coverage rates directly translate to a commensurate reduction in the number of instructions needing book-keeping resources at those pipeline stages for which fusion takes place, be that fetch, rename, schedule, or commit, as well as a reduction in the number of values needing to be written to physical registers and the number of instructions requiring instruction and issue queue entries.

Micro-op and macro fusion. Two complementary instruction fusion techniques found in modern processors are *micro-op* and *macro fusion*. As real-world examples of instruction fusion, micro-op and macro fusion provide this dissertation with a useful backdrop for introducing and explaining mini-graph processing.

Micro-op fusion was introduced in Intel's Pentium M [38, 51], which fuses load/execute and store-address/store-data micro-op pairs into expanded micro-ops. Examples of each of these pairs are shown in Figure 1.2a-b. Load/execute and store instructions are x86/macro instructions. Instead of decoding the macro instruction

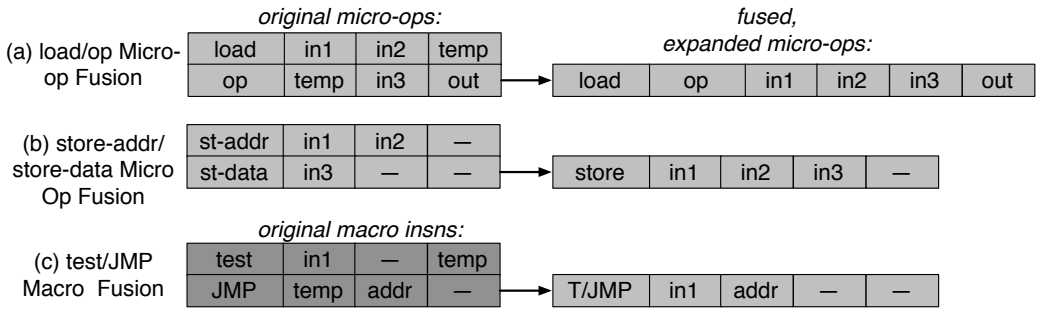


Figure 1.2: Representation of Micro-op and Macro Fusion.

into two micro-ops (shown on the left), micro-op fusion decodes them into a single, expanded micro-op (shown on the right). The expanded micro-op is a superset of its original micro-op constituents.

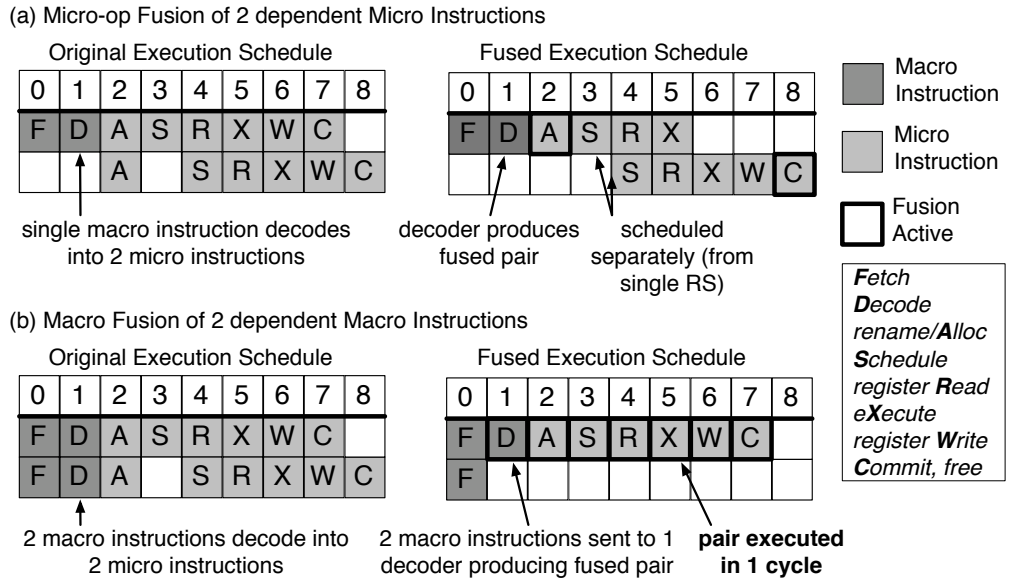


Figure 1.3: Timing of Micro-op and Macro Fusion.

An example of micro-op fusion in the pipeline is shown in Figure 1.3a. On the left is the execution schedule of the original dependent micro-ops. On the right is the execution schedule under micro-op fusion. Micro-op fusion decodes the macro instruction into a single micro-op, which occupies a single reorder buffer (ROB) and

issue queue entry. The ROB and the issue queue are amplified as a consequence. Micro-op fusion also reduces the number of micro-ops that are renamed and committed. Finally, by reducing the number of x86 instructions that decode into multiple micro-ops—such instructions decode on the lone complex decoder—micro-op fusion also amplifies the decode bandwidth. The constituents of the fused micro-op are still scheduled separately—one directed to the load unit, the other to a simple ALU—and therefore do not amplify issue or execution bandwidth [38].

Macro fusion was introduced in Intel's Core processor [72, 102]. Macro fusion fuses test or compare instructions to conditional branches. An example of the original micro-ops and the fused, expanded micro-op is shown in Figure 1.2c. An example of macro fusion in the pipeline is shown in Figure 1.3b. Whereas micro-op fusion fuses micro-ops after the decode stage, macro fusion fuses x86 macro instructions prior to the decode stage. Macro fusion further extends the benefit of fusion to execution latency reduction; as shown in the execution schedule, the fused instructions are executed in a single cycle on a modified branch unit. Because they are executed on a single functional unit, they are also issued atomically, unlike fused micro-ops.

Intel estimates that micro-op fusion reduces the number of micro-ops handled by the out-of-order logic by more than 10% [38]. In other words, it has 10% coverage. This assumes, however, that store instructions are already split in two. For those ISAs that do not make this assumption—like the Alpha ISA used to measure the coverage of mini-graphs—micro-op fusion would result in less coverage. Some estimates show that adding macro fusion to micro-op fusion increases coverage to just over 15%. The typical performance improvement offered by micro-op fusion for integer benchmarks is approximately 5%. Performance estimates for macro fusion have not yet been made publicly available at the time of this dissertation, but would likely create a total improvement between 7% and 10%. Although relatively simple to implement and moderately effective at reducing bookkeeping costs in the processor, both techniques are limited. First, neither of them offer fusion benefits

that extend to the entire pipeline. Second, both forms of fusion apply to only two specific instruction pairings (load-and-execute and store-address-and-store-data for micro-op fusion, test/compare-branch for macro fusion).

1.3 Mini-Graph Processing

Mini-graph processing is an aggressive instruction fusion technique [12, 14]. In the context of micro-op and macro fusion, mini-graph processing can be thought of as a logical extension to both techniques.

Mini-graph processing is aggressive with respect to amplification, which it maximizes in two respects. First, it extends the benefits of fusion to more instruction combinations than the few pairing options available in micro-op and macro fusion. Second, it expands the scope of instruction fusion, extending its benefits to more pipeline stages and structures. At the same time, mini-graph processing remains conservative with respect to the changes it requires to the pipeline and ISA. The operating system need only be aware of mini-graph processing in order to disable it—a debugging aid, but not a necessity for correctness (see Section 2.2).

Moderate coverage rates provide incremental IPC improvements to the extent that amplified resources are performance limiters. With sufficiently high coverage rates, however, mini-graph processing can serve as a replacement for superscalar width and window size. To give a concrete example, if mini-graphs can achieve coverage/amplification rates of 33%, then a conventional 4-wide dynamically scheduled superscalar machine with a 128 entry reorder buffer and a 64 entry issue queue can be replaced with a 3-wide superscalar with a 96 entry reorder buffer and 48 entry issue queue *plus* mini-graph support. This goal is accomplished in the following five steps.

1.3.1 Maximizing Coverage

Micro-op and macro fusion both focus on fusing pre-defined operation pairs. Mini-graph processing supports fusion of larger instruction aggregates (called *mini-graphs*) and also application-specific fusion. These capabilities require three pieces of innovation. The first is the separation of the mini-graph interface from the mini-graph operations. The second is an on-chip structure, the Mini-Graph Table, that stores the internal definition of each mini-graph's operations. The third is an encoding mechanism, called *annotated outlining*, whereby mini-graphs can be encoded into a binary and subsequently used to program the Mini-Graph Table at runtime.

Instruction = Handle + Template. Mini-graphs are manipulated at two levels: the handle and the template. The interface is represented by a *handle*, a quasi-instruction that the mini-graph processor manipulates while the mini-graph travels down the pipeline. The mini-graph handle has three components: an opcode, register specifiers, and an immediate field. Mini-graphs use a reserved opcode `mg`, which can be any opcode (or several opcodes) that would be interpreted as a `nop` by a non-mini-graph processor. The handle is meaningful only to a mini-graph enabled processor.

The handle specifies three register inputs and an output. The four register names explicit in a handle are the mini-graph's interface registers which define its external dependences. These register names (or their renamed versions) are needed at renaming, scheduling, register read, register write, retirement, and mis-speculation recovery; stages where only the handle is available, not the complete mini-graph. Finally, the handle has an immediate field, the `MGID`, that links the handle to its internal definition. The `MGID` connects the mini-graph interface to its definition, specifying which mini-graph template should be invoked at execute. In this respect, it is similar to the parameterizable instruction used to exploit hardware-programmable functional units on a PRISC machine [83]. Using the `MGID` rather than, say, the address of the handle, allows two different, static locations of code to invoke the same mini-graph

template.

The *template* encodes the mini-graphs's internal definition. A mini-graph template specifies the exact constituent operations (opcodes and immediates) as well as the internal register dataflow of the mini-graph. Each template is assigned a unique identifier which is stored in the handle's `MGID` field. Because the template specifies internal register dataflow without using actual register names, a single template can be used to specify mini-graphs in multiple static locations in the code.

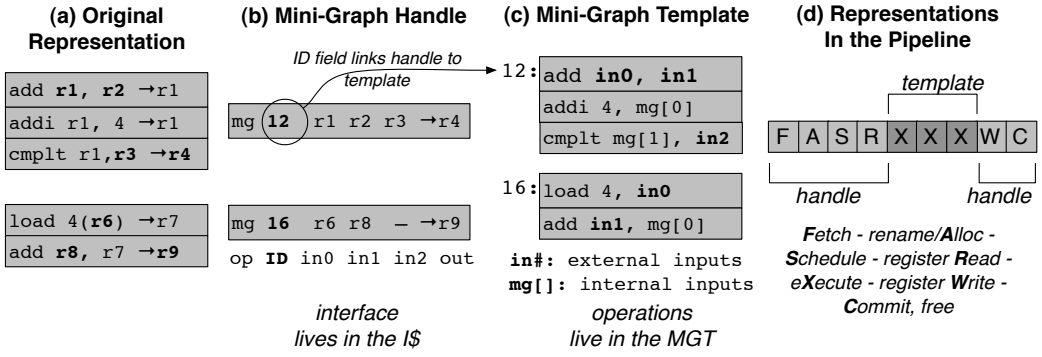


Figure 1.4: Representing Mini-Graphs: Handles and Templates.

Mini-graph representation example. Figure 1.4a shows two instruction sequences. Figure 1.4b shows the handle that summarizes each sequence. The `MGID`, `12` and `16`, indexes the corresponding definitions of the mini-graphs. Figure 1.4c shows the template for each mini-graph. Note, this figure is logical; the organization and contents of the actual structure that holds templates are described later. External input registers are denoted with a combination of `in` and their index in the handle (`0,1,2`) while interior values are denoted using `mg[]` indexed with the mini-graph instruction that creates them. For example, the third mini-graph constituent of `mg12`, `cmplt r1, r3, → r4` is represented as `cmplt mg[1], in2` where `in2` is interface input register `r3` and `mg[1]` is output of the second mini-graph instruction. Interior values such as `mg[1]` are provably transient by static analysis and do not need to be stored in registers at runtime.

Finally, Figure 1.4d illustrates which representations are used at which pipeline stages. Whereas the interface is needed throughout the pipeline, the mini-graph definition is needed only at the schedule and execution stages. At schedule, a summary of mini-graph resource needs (not shown) directs resource reservations required to schedule the handle; at execution, the template drives constituent execution.

The Mini-Graph Table (MGT). Mini-graphs exploit the most common operational idioms found for each program. Fused pairs supported by micro-op fusion (*e.g.*, load-and-execute) require permanent changes to the decoder and scheduler and are therefore conservatively general so as to have utility across all possible programs. Mini-graphs, on the other hand, can be defined for a single program. The enabling mechanism for this flexibility is the MGT.

The MGT factors fused execution information out of the decoder and issue queue and into a separate structure. This means that the decoder is modified only to recognize the class of instructions known as mini-graphs. Furthermore, this allows a conventional issue queue and conventional scheduler to drive the execution of arbitrary aggregates.

The MGT holds all the template definitions for every mini-graph handle occurring in a particular program binary. During execution, a mini-graph processor invokes the MGT to drive a cycle-by-cycle execution of the constituent operations, micro-code style. The handle has an immediate field called the `MGID`, which is serves as the index into the MGT. Using an `MGID` field to index into the MGT enables multiple dynamic locations of semantically identical instruction sequences to use a single MGT entry. The MGT is implemented as a cache. MGT size is not part of the architected state of a mini-graph processor, but informing the software tool that prepares the mini-graph binary with the MGT size can improve performance (see Section 2.6).

Annotated outlining. *Annotated outlining* is a novel encoding scheme used to transform a standard binary into a mini-graph binary. The process is shown in Figure 1.5. First, the instructions that form the body of the mini-graph are made

contiguous and prepended with the mini-graph handle. This extended sequence is then "outlined" from the code (as opposed to "inlined") using a pair of jumps (Figure 1.5b). As shown in Figure 1.5c, the instruction cache fill path recognizes the handle and places it in the instruction cache. The mini-graph constituent operations are pre-processed into template form and diverted to the MGT at the index corresponding to the handle's `MGID`.

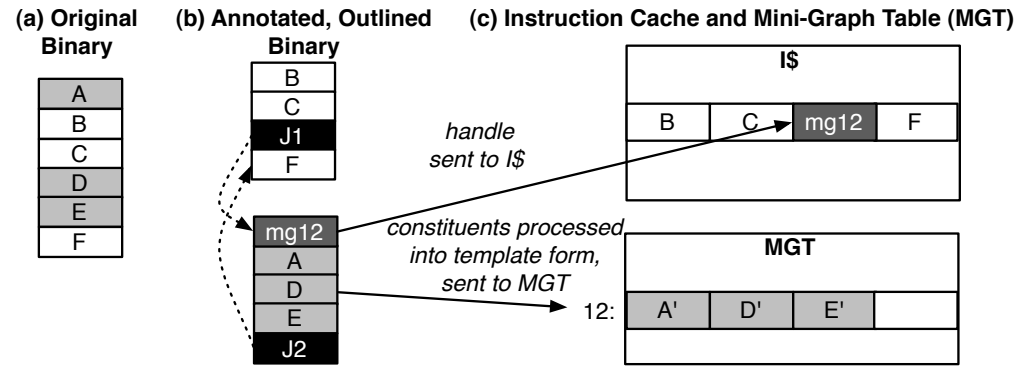


Figure 1.5: Creating an Annotated, Outlined Binary

1.3.2 Maximizing Amplification

Mini-graph processing not only increases coverage to arbitrary combinations of instructions, but it also extends the benefits of amplification to new pipeline stages and structures. Micro-op and macro fusion amplify decode, rename, and commit bandwidth, and issue queue and reorder buffer capacity. Because their target processors perform architectural register and ROB-style renaming, amplifying ROB capacity has the same effect that amplifying register file capacity has on a processor that performs physical register renaming. Mini-graphs extend bandwidth amplification to fetch, register read/write, and integer execution and capacity amplification to the instruction cache. Doing this requires three pieces of innovation.

Amplifying instruction cache capacity and fetch bandwidth. Instruction

cache capacity and fetch bandwidth amplification is one of many benefits of annotated outlining. This benefit comes as the simple consequence of representing an entire mini-graph as a single instruction (*the handle*) in the instruction cache.

Amplifying the register file. Without static analysis to guarantee the transience of values, micro-op and macro fusion can target instruction pairs that have at most one register output. Consecutive load and op x86 macro instructions cannot be fused because they could have more than one output; load and op instructions from the *same* macro instruction can only have a single output because the value passed from the load to the op is deemed transient by the compiler and therefore never assigned to a register. Hence, the number of values written and communicated globally does not change as a result of micro-op fusion. Mini-graph processing targets instruction groups whose constituents may produce multiple values, so long as all but one of these values are statically, provably *transient*—existing only within the mini-graph. Value communication is orchestrated by the MGT using bypasses and latches without actual registers, amplifying register file capacity and register read and write bandwidth. Transience is determined through static liveness analysis, and—to guarantee transience—mini-graphs are atomic.

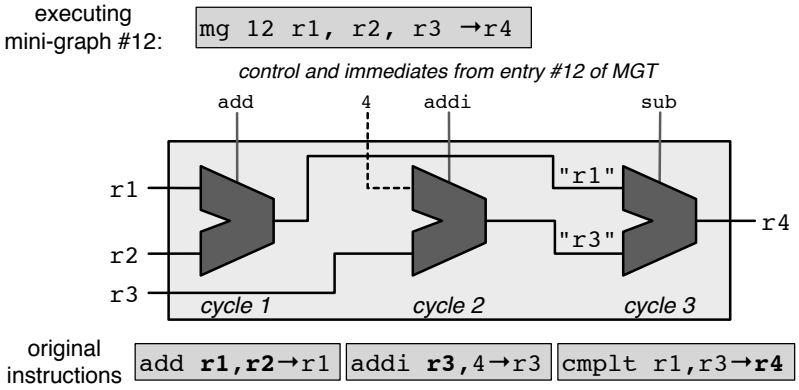


Figure 1.6: 3-input, 3-stage ALU Pipeline

Amplifying execution with ALU Pipelines. Mini-graph processors further exploit mini-graph interior value transience by executing chains of integer constituents on a new, multi-cycle functional unit called an *ALU Pipeline*: a single-entry, single-exit chain of ALUs. An ALU pipeline is essentially a multi-cycle functional unit (like a multiplier). Figure 1.6 shows a 3-input, 3-stage ALU pipeline executing the previously shown `mg12`. In the case of `mg12`, the inputs arrive at the beginning of cycle 1 and the output is ready at the end of cycle 3. At each cycle, each ALU receives the requisite immediates and control signals from the MGT. Transient values are propagated via the forward-only operand network between ALUs and do not access the register file. The execution of multiple mini-graphs may be pipelined across this new functional unit.

ALU Pipelines play a critical amplification role. Because execute is the only pipeline stage that manipulates constituents rather than handles, it is the only pipeline stage whose bandwidth is not naturally amplified by fusion. ALU Pipelines prevent execution from becoming a new bandwidth bottleneck by adding integer ALU execution bandwidth (which is relatively inexpensive) without commensurate increases in global bypass, register file, and scheduling bandwidths (which are more expensive). Because singleton instructions can also execute on ALU pipelines with no performance penalty, a mini-graph processor simply replaces some of its ALUs with ALU pipelines.

Putting it all together. Figure 1.7a-b shows the execution schedules of micro-op and macro fusion (unchanged from Figure 1.3). Figure 1.7c shows the same for mini-graph processing. Unlike the micro-op and macro fusion, mini-graphs are fused for the entire pipeline: from fetch to commit, excepting execute. Processing mini-graphs as handles at almost all stages maximizes amplification while minimizing the number of changes required to the pipeline; only the instruction cache fill, schedule, and execute stages are made mini-graph aware.

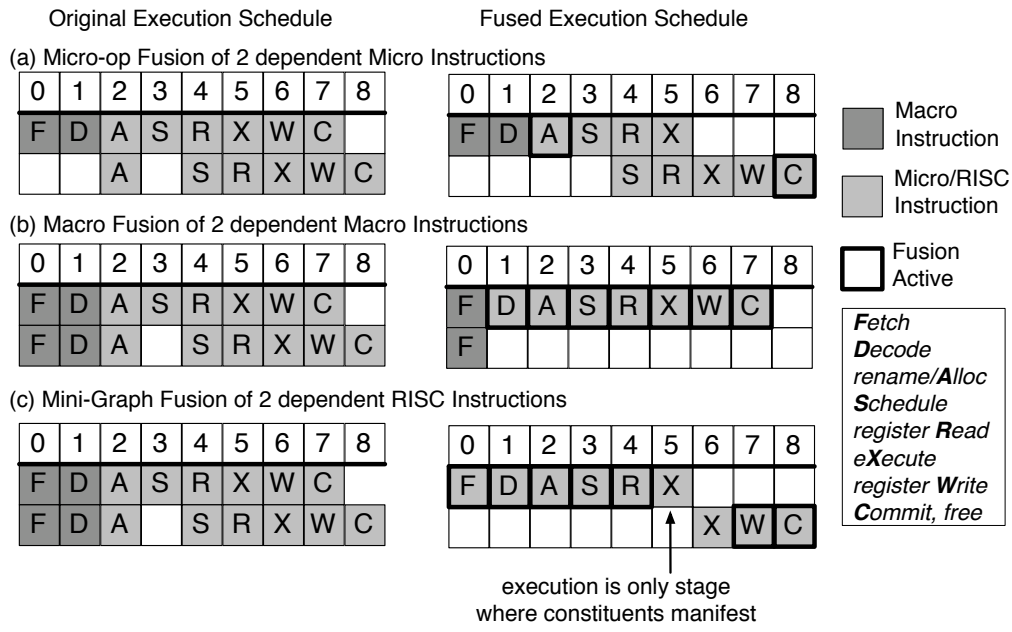


Figure 1.7: Mini-Graph Fusion.

1.3.3 Supporting Limited Latency Reduction

There are actually two approaches to instruction fusion. Mini-graphs are an amplification-oriented form of fusion; they focus on amortizing interface coordination. Most forms of fusion are actually latency-oriented, focusing on reducing the execution latency of common operation chains (or more generally graphs). Latency-oriented fusion can actually shorten the height of a program's dataflow graph.

Mini-graphs can incorporate the benefits of execution latency reduction by supporting the pair-wise collapsing of ALU pipeline stages. To illustrate, if two ALU pipeline stages could execute in a single cycle, cycles 5 and 6 of Figure 1.7 would collapse into a single cycle, as was achieved for macro fusion in Figure 1.3b. Furthermore, mini-graph processing can apply latency reduction to arbitrary pairs of arithmetic/logical operations, not just the test/compare-branch pairs of macro fusion.

1.3.4 Minimizing Design Impact

Although aggressive with respect to amplification, mini-graph processing is designed to be conservative in terms of hardware and software design changes. To minimize pipeline modifications, mini-graphs are constrained to have the interfaces of RISC singleton instructions. These constraints are detected statically by a software tool that identifies legal mini-graphs. Most importantly, mini-graphs are constrained to be atomic, have at most three register inputs and one register output, and to perform at most one memory operation. This last constraint preserves instruction-granularity handling of memory operations, including memory related exception handling. Existing mechanisms like branch prediction, memory disambiguation, and load-store forwarding remain unchanged in the presence of mini-graphs; mechanisms that use instruction PC (*e.g.*, branch prediction) can use handle PC instead. Annotated outlining makes a minor ISA extension that requires only one or two new opcodes; annotated outlining naturally supports functional compatibility across different mini-graph and non-mini-graph processors. Because the binary contains the original instruction sequences, these can be retrieved to aid in debugging or to handle rare and difficult exceptions. Mini-graphs do not need to be explicitly virtualized and require OS support only if disabling mini-graphs is desired.

1.3.5 Providing Robust Performance

Fused micro-ops occupy a single issue queue slot, but they issue separately. Mini-graph constituents, on the other hand, are issued atomically. The benefit is obvious. Whereas micro-op fusion amplifies only the issue queue, mini-graph processing amplifies both the issue queue and issue bandwidth. There is, however, a cost. Requiring mini-graph constituents to issue atomically creates the possibility of *serialization*, the introduction of new program dependences. Because a mini-graph cannot issue until all of its external inputs are ready, data dependences exist between all producers of mini-graph inputs and the first operation in the mini-graph.

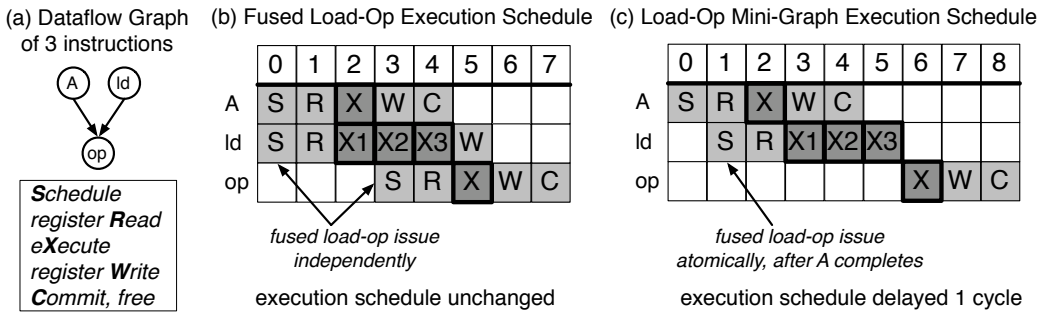


Figure 1.8: Serialization Effects in Mini-Graphs

Figure 1.8 shows an example with three instructions `A`, `ld`, and `op`, shown in Figure 1.8a. Figure 1.8b shows the execution schedule of these three instructions, with the load-op pair fused by micro-op fusion. Figure 1.8c shows the execution schedule of the same three instructions, with the load-op pair as a mini-graph. (The pipelines begin with the schedule stage for simplicity.) Whereas the micro-op pair are scheduled independently, allowing the load to execute in parallel with instruction `A`, the mini-graph is scheduled atomically, requiring the load to wait for instruction `A` to complete before it can be executed. The result is a 1 cycle delay in the execution schedule. The load waits for a value that it does not actually depend on; this delays not only the load but also the `op` that follows it.

Small increases in execution latency caused by serialization can result in overall program slowdowns. This dissertation provides an extensive examination of the problem of serialization and serialization-aware fusion in the context of mini-graphs. Selecting mini-graphs aggressively (*i.e.*, ignoring serialization) maximizes mini-graph coverage but the associated performance loss for many programs outweighs the benefits of amplification in the first place. Selecting mini-graphs conservatively (*i.e.*, prohibiting serialization by static analysis) avoids performance loss but produces little amplification. Most instances of serialization are benign (unmanifested or unimportant). This dissertation develops three serialization-aware selection schemes that identify and reject mini-graphs with *harmful serialization only.*

1.4 Results Summary

Mini-graph processing is a low-cost, low-area, low-design-impact substitute for superscalar width and window capacity. Current processors are converging upon the 4-wide issue core as a performance-efficiency design sweet-spot [106]. Given the goal of designing low-power, low-area cores that still maintain high single-thread performance, the question is whether the coverage offered by mini-graphs is sufficient to achieve the performance of the 4-wide core with a mini-graph-enabled 3-wide core. Answering this question requires a study of both the coverage and performance of mini-graph processing. Specifically, a performance comparison is made between a 3-wide mini-graph processor and a 4-wide non-mini-graph processor. Figure 1.9 details both the 3-wide and 4-wide configurations, and the mini-graph support. Both the 3-wide and the 4-wide configurations are tuned to the performance "knee" for both issue queue entries (20 vs. 30) and physical registers (120 vs. 144).

Parameter	Configuration			
Memory System	32KB, 2-way/4-way associative 3-cycle access data/instruction 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	16 stages: 1 predict, 3 instruction cache, 1 decode, 2 rename, 1 schedule, 2 register read, 1 execute, 1 register write, 3 replay, 1 commit			
Instruction Window	128-entry ROB, 64-entry load queue, 32-entry store queue. Loads are scheduled aggressively using a Store Vulnerability Window. Memory ordering violations flush the pipeline. Cache miss replays are modeled.			
Processor Width	fetch/issue/commit width	issue queue size	register file size	Scheulder Issues per cycle: (maximum)
4-wide	4-wide	30 entry	144 pregs	4 int / 1fp / 2 load / 1 store
3-wide	3-wide	20 entry	120 pregs	3 int / 1fp / 2 load / 1 store
Mini-Graphs	Maximum 5 instructions, 7 cycles, 3 integer constituents, 1 load or store, 1 terminal control instruction. Scheduler issues at most 2 mini-graphs per cycle, at most 1 integer mini-graph per cycle. MGT SIZE: 512-entries, 7 banks. 2 3-stage ALU Pipelines replace standard ALUs.			

Figure 1.9: Simulation Configuration for 3-wide and 4-wide processors.

Methodology. This dissertation studies mini-graphs in the context of user-level code, the Alpha AXP ISA, and processors with unified physical register files [36, 43, 110]. Mini-graph processing in the context of non-Alpha ISAs is discussed

in Section 2.7; mini-graphs in the context of alternative microarchitectures is discussed in Section 2.8. Mini-graph binaries were created from 78 Alpha binaries across four benchmark suites: SPECint2000 (SPEC), MediaBench [60], CommBench [108], and MiBench [42]. The original binaries were 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. They are included to show the applicability of mini-graphs to different kinds of codes.

The timing simulation infrastructure uses the SimpleScalar 3.0 Alpha AXP instruction definition and system call modules to model a dynamically scheduled superscalar processor. The simulator uses cycle level simulation to generate IPCs, but does not explicitly account for circuit delay, frequency, or power. Performance results are quantitative insofar as the cycles required to execute each instruction are faithfully modeled by each pipeline stage. The performance results do not, however, account for possible frequency changes. Power and frequency results are strictly qualitative; particular processor configurations are determined to consume more or less power based on relative capacities and bandwidths of particular critical on-ship structures.

This section presents results for both coverage and performance, as shown in Figure 1.10. Much of the data in this dissertation is displayed using S-curve graphs. Each line represents an experiment in which all programs are sorted from worst to best; hashes mark each program. In the same graph, each experiment is sorted independently so that the same horizontal point may correspond to different programs in different experiments. To illustrate, the large diamond in each graph shows the results for the SPEC benchmark *twolf*. On a 4-wide processor, *twolf* sees a

10.6% performance improvement over a 3-wide processor. On a mini-graph processor, *twolf* has a 30% coverage rate and sees a 15.6% performance improvement over a 3-wide processor. S-curves effectively display trends and medians for large numbers of benchmarks and prevent outliers from hiding in averages.

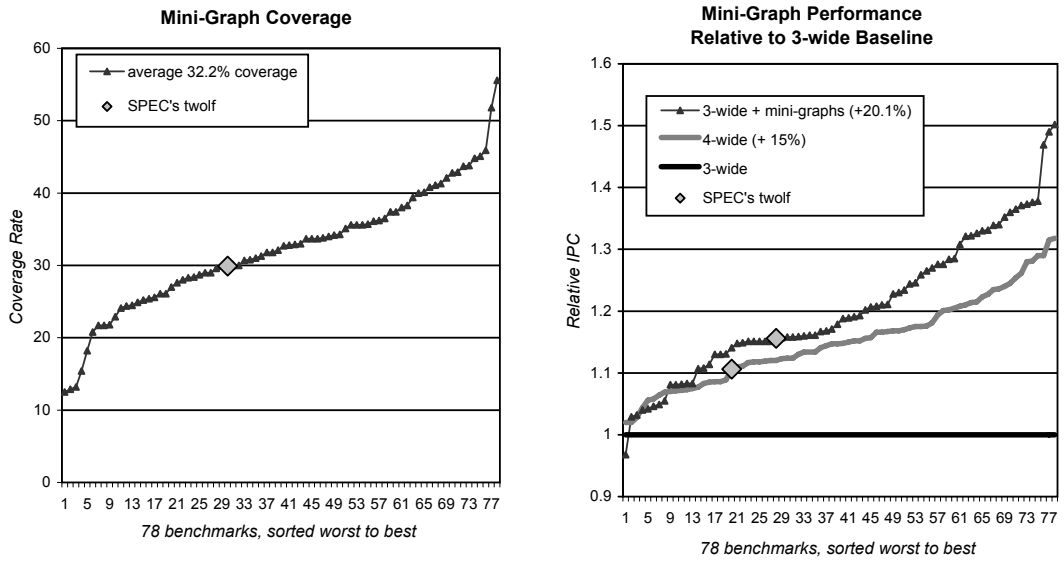


Figure 1.10: Mini-Graph Coverage and Performance. Left: Coverage of mini-graphs across 78 benchmarks. Right: Comparison of 3-wide mini-graph processor and 4-wide processor. Relative to a 3-wide processor. Mini-graphs in this graph are chosen using the best performing serialization-aware selection algorithm, $\text{Slack}_{\text{Profile}}$ (see Chapter 3).

Coverage. Coverage varies based on the mini-graph definition (how many and what kinds of instructions can be incorporated into mini-graphs). The graph on the left of Figure 1.10 shows coverage rates for mini-graphs with as many as three integer operations and at most five total constituents.

Coverage across 78 benchmarks is approximately 32%. Structures and pipeline stages which process handles are effectively one third larger. This is achieved without physically increasing capacity or bandwidths. The next graph compares the performance of a mini-graph processor with a non-mini-graph processor that *is* physically larger by this same (one-third) amount.

Performance. The performance effects of mini-graph processing can be observed by comparing a less-provisioned (*i.e.*, 3-wide) processor with mini-graph capabilities to a more-provisioned (*i.e.*, 4-wide) processor without mini-graphs. This comparison is shown in the graph on the right of Figure 1.10. Performance is relative to a 3-wide, non-mini-graph baseline. The 3-wide mini-graph processor outperforms the 4-wide non-mini-graph processor in almost every case; the former shows a 20% performance improvement over a baseline 3-wide machine, the latter only a 15% improvement.

The mini-graphs in this experiment are selected using the best performing serialization-aware selection algorithm, $\text{Slack}_{\text{Profile}}$, which is introduced in Chapter 3. It is important to note that without this algorithm, the performance improvement is both half that of the 4-wide machine and significantly less robust; approximately 25% of benchmarks would see a performance *loss* over the 3-wide baseline.

A 3-wide mini-graph processor not only matches the performance of a 4-wide non-mini-graph processor, but it does so with smaller superscalar structures and bandwidths. (These data do not assume latency reduction in the ALU Pipeline; each operation executes in a full cycle.) The main hardware cost of mini-graph processing is the MGT, whose capacity, bandwidth, and support for banking (as explained in the next chapter) makes it inexpensive, relative to the cost of adding entries and ports to existing structures (*e.g.*, register file, issue queue) or increasing the bandwidth of the entire processor by one-third. Mini-graph processing can, in fact, act as a robust, less expensive replacement for some of the expensive bookkeeping machinery of a high-performance processor.

1.5 Contributions

This dissertation makes several contributions. Specifically, it:

- Introduces the concept of mini-graphs processing, a form of instruction fusion

that focuses on pipeline capacity and bandwidth amplification.

- Describes a novel microarchitecture for processing mini-graphs that requires only small modifications over existing superscalar designs. The key components of the implementation include the Mini-Graph Table, ALU Pipelines, and an outlining-based ISA extension facility.

- Presents the first extensive discussion of the problem of serialization. Introduces both external and internal serialization as well as the conditions under which each can affect performance. Develops and evaluates several serialization-aware mini-graph pruning algorithms. Compares these algorithms to each other as well as to naive and exhaustive methods.

- Presents a simulation-driven performance evaluation of the complete mini-graph system, showing that the addition of mini-graph processing allows a dynamically scheduled 3-wide superscalar processor to match the IPC of a 4-wide superscalar machine.

Previously published work on mini-graph processing include a 2004 publication in the 37th International Symposium on Microarchitecture [12], a 2006 publication in the 39th International Symposium on Microarchitecture [14], and a 2006 technical report from the University of Pennsylvania [13]. The first publication introduces the concept of mini-graph processing including the MGT and ALU Pipelines, discusses the coverage across different possible mini-graph definitions, introduces the concept of serialization, and shows initial performance benefits for various mini-graph processor configurations. The second publication briefly discusses the mini-graph encoding scheme but focuses primarily on serialization. It introduces several pruning algorithms designed to overcome the performance penalties caused by serialization. The technical report best describes the outlining-based ISA extension facility used to create a mini-graph binary; this encoding differs from the DISE-based encoding [26] of the 2004 publication. These three publications correspond to Chapter 2, Chapter

3, and Section 2.2, respectively. That said, this dissertation—particulary Chapter 2—presents a more extensive discussion of mini-graph processing than is present in any of the publications.

This dissertation is organized as follows. Chapter 2 describes the architecture and microarchitecture of a mini-graph processor, detailing how mini-graphs are encoded in a program binary and how they are processed as the program executes. Chapter 3 discusses how mini-graphs are identified statically and how they can be chosen to maximize both resource amplification and performance. This section focuses on minimizing performance penalties associated with serialization. Chapter 4 presents a simulation-based timing evaluation of mini-graph processing. Chapters 5 and 6 discuss related works and conclusions, respectively.

Chapter 2

Mini-Graph Architecture

Mini-graph processing is a unique form of instruction fusion that targets bandwidth and capacity amplification throughout the entire pipeline, from fetch to commit. By performing certain actions once per aggregate instead of on a per-instruction basis, structure bandwidth and capacity is allocated to other instructions, creating an amplification effect. This wholesale amplification enables either improved IPC throughput at a fixed resource point or, alternatively, fixed (or better) IPC with fewer resources.

This dissertation studies mini-graphs in the context of user-level code written in the Alpha ISA. The assumed microarchitecture is similar to an unclustered Alpha 21264, specifically one that implements register renaming using a unified physical register file (as opposed to an architectural register file and a value-based ROB), that has a unified scheduler for integer and memory operations and a separate scheduler for floating-point operations.

Mini-graphs are designed to maximize amplification both from a dynamic instruction standpoint and from the standpoint of number of structures and pipeline stages amplified. At the same time, they are designed to minimize impact on the microarchitecture—specifically the number of structures that are made mini-graph aware—the ISA, and the operating system. In other words, mini-graphs maximize

their amplification impact while minimizing their implementation costs.

This chapter begins with a brief overview of the structural changes required to support mini-graph processing. Figure 2.1 depicts a high level diagram of a basic out-of-order processor. The unshaded parts of the figure show the structures that are unmodified in a mini-graph processor: the instruction and data caches, the branch predictor, the floating-point units, as well as more complicated entities such as the decoder, the load/store queue, register renaming, as well as the register and memory schedulers. The shaded parts of the figure show the structures that are added in order to support mini-graph processing: the Mini-Graph Table, the Mini-Graph Pre-Processor, and ALU Pipelines. Structures outlined in bold (issue queue, functional units) are modified from the basic processor in order to support mini-graph processing. The bold lines show new or modified paths between the structures themselves. Each shaded structure or bold line is labeled with the section number in which this support is discussed.

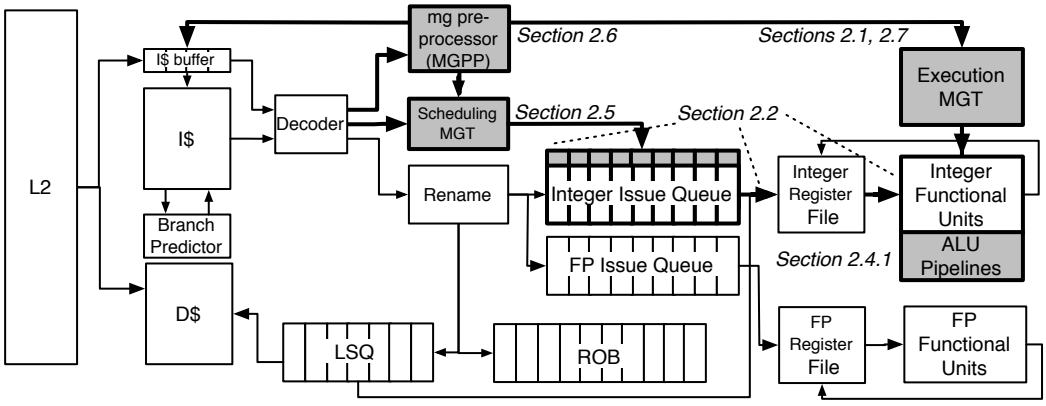


Figure 2.1: Architectural Changes

Mini-Graph Table (MGT). Mini-graph templates reside in an on-chip structure called the *Mini-Graph Table* (MGT). The MGT is an on-chip cache for mini-graph templates—each entry holds the resource needs and definition of one mini-graph template, indexed by `MGID`. Mini-graph templates are entered into the MGT

on a per-program basis; this allows each program to have its own unique set of templates. Mini-graph templates do not need to be shared across applications.

The MGT is divided into two parts: the Scheduling MGT and the Execution MGT, shown in Figure 2.2a. The *Scheduling MGT* (see Section 2.4) summarizes the resource needs of each mini-graph template. Mini-graph scheduling information is accessed at dispatch and then used by the scheduler to coordinate resource reservations. For example, the Scheduling MGT informs the scheduler that the mini-graph `mg12` in Figure 2.2a has an execution latency of 3 cycles and therefore needs a correspondingly delayed reservation of a writeback port.

The *Execution MGT* holds the per-constituent template information and drives the cycle-by-cycle execution of the mini-graph's constituent operations, micro-code style. Each entry contains an opcode, an immediate, and a group of control bits. The opcode and immediates are found in the original non-mini-graph constituents. The control bits encode the dataflow of the mini-graphs, driving the input muxes to functional units and the output muxes of the ALU Pipeline (explained next).

The Execution MGT is both banked and pipelined; n banks are required to support a mini-graph with an execution latency of n cycles. Figure 2.2, for example, shows and Execution MGT where $n = 3$. The first bank controls the execution of the first mini-graph constituent during the first cycle of execution of the mini-graph. If a mini-graph does not execute a new constituent at a particular cycle, the corresponding bank's entry remains unused. m read ports per bank support the continuous pipelining of m mini-graphs per cycle. With this configuration, the MGT is free of structural hazards. Assuming a 6-bit opcode, a 16-bit immediate, two 3-bit mux controllers (for selecting inputs), 1 bit indicating an output, and remaining bits to indicate which functional unit a constituent executes on, each MGT constituent requires 32 bits, *i.e.*, 4 bytes. A 512-entry MGT, then, requires 2KB per MGT bank. A 7-bank MGT is 14KB.

The configuration of the MGT is one of the primary levers that trades mini-graph coverage with mini-graph implementation costs. Adding entries to the MGT either supports fewer MGT misses and improves performance or supports more mini-graph templates and consequently increases coverage. Adding banks to the MGT supports longer mini-graphs and consequently increases coverage. Adding more ports to each bank supports the execution of more mini-graphs per cycle and consequently increases mini-graph throughput. However, increasing the number of MGT entries, banks, or ports all increases the cost of mini-graph processing by increasing the size and power consumption of the MGT itself.

ALU Pipelines. Execution is the only pipeline stage not amplified by mini-graph processing. To prevent execution from becoming a new 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 (see Figure 2.2b). An ALU Pipeline is essentially a multi-cycle functional unit (like a multiplier) on which integer mini-graphs execute. ALU Pipelines add ALU execution bandwidth without requiring matching increases in register file and bypass bandwidths.

Like the MGT, the configuration of the ALU Pipeline is another lever used to trade coverage with implementation costs. ALU Pipelines with more stages support longer mini-graphs that achieve more coverage. Adding an ALU Pipeline stage incurs the same area and power costs of adding a standard ALU, but none of the bypass complexity of actually increasing the execution width of the processor. The ALU Pipeline support also affects performance. Having multiple ALU Pipelines will not improve coverage, but it will improve throughput. Finally, the cost of an interlock-collapsing ALU Pipeline comes with the benefit of reducing the execution latency of the mini-graph and potentially shortening the height of a program's dataflow.

MGT and ALU Pipeline example. Figure 2.2 shows an MGT that supports integer mini-graphs of up to three instructions long connected to its corresponding ALU Pipeline (of 3 ALUs). The MGT is indexed by the `MGID` field of the mini-graph

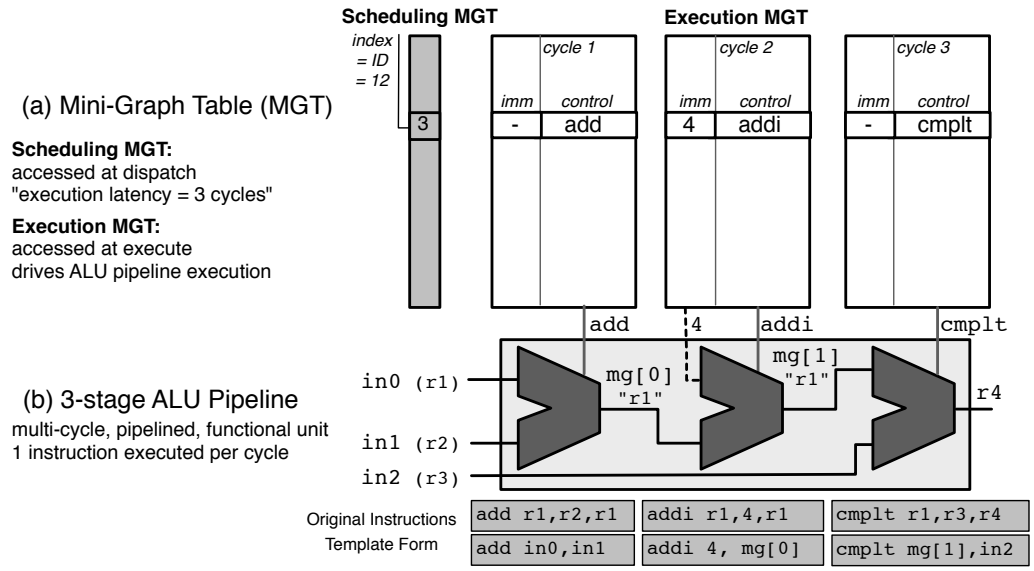


Figure 2.2: Mini-Graph Table and ALU Pipeline.

handle (in this case, `12`). At dispatch, the Scheduling MGT is accessed to determine the execution latency of the mini-graph; this information extends the scheduler itself and is used in the next cycle. The Execution MGT contains the necessary immediate inputs, opcodes, and input selectors (not shown) in order to drive the execution of the mini-graph on the ALU Pipeline. For simplicity, the ALU Pipeline in Figure 2.2b shows only paths that are used by this particular mini-graph. Scheduling and executing integer-memory mini-graphs is slightly more complex and is discussed in Section 2.3.

Integer-memory mini-graphs. The final lever used to trade coverage and cost is the mini-graph definition itself. This dissertation focuses on two types of mini-graphs: *integer mini-graphs*, which contain only single-cycle ALU operations and *integer-memory mini-graphs* which can also contain loads, stores, and control transfers. Because they behave almost identically to standard multi-cycle instructions, integer mini-graphs require little support. Unfortunately, they also have an average 14% coverage rate—not enough to compete with an additional superscalar slot.

Because they incorporate more types of instructions, integer-memory mini-graphs achieve coverage rates of approximately 35%. This amplification benefit comes at an implementation cost. Three-cycle loads make inefficient use of the MGT which is banked by cycle, not instructions. Integer-memory mini-graphs also fundamentally require coordination of multiple functional units; ALU Pipelines cannot easily be extended to support memory or control instructions. This coordination impacts both the scheduler and the bypass network, although careful design can minimize this impact.

The mini-graphs supported in this dissertation is as follows. Integer mini-graphs (or the integer constituents of an integer-memory mini-graph) are executed on 3-stage ALU Pipelines. Integer-memory mini-graphs may contain up to three consecutive integer operations, a maximum of one loads or stores, and a single control instruction—up to 7 cycles of execution. Load latency is assumed to be three cycles. Two mini-graphs may issue per cycle, but only one of these may be an integer-memory mini-graph.

This chapter reviews the basics of mini-graphs and mini-graph processing with these design goals in mind. It describes all the necessary components of a mini-graph architecture. Section 2.1 details mini-graph criteria. Section 2.2 explains how mini-graphs, once identified in a program, are encoded in a program binary. Section 2.3 introduces the microarchitectural support necessary to execute mini-graphs. Section 2.4 describes mini-graph scheduling. Section 2.5 discusses the Mini-Graph Pre-Processor whereas Section 2.6 provides further details about the MGT. Additional attention is needed in order to select mini-graphs that guarantee robust performance. Chapter 3 discusses selection for robustness and high performance.

2.1 Mini-Graph Criteria

Mini-graphs are aggregates with the external appearance of singleton RISC instructions. The RISC singleton interface makes mini-graphs appropriate for superscalar processors which rely on simple book-keeping units to implement register renaming and dynamic scheduling. Mini-graph handles are designed to look and behave as any other singleton instruction and require as little special handling as possible.

Mini-graphs are first and foremost *atomic* – constituent operations are executed on an all-or-none basis, and internal mini-graph state is unobservable by external instructions. Atomicity is the key to amplification; pipeline stages and superscalar structures that manipulate instructions do so at a mini-graph (not constituent) granularity. Atomicity also allows register communication that is "interior" to a mini-graph to take place without actual registers. Provably transient values live only in the bypass network. This amplifies both the capacity of the physical register file as well as the bandwidths of all stages that manipulate either register names (rename/register-allocate and commit/register-free) or values (register read and write).

Atomicity constrains mini-graphs to reside within atomic instruction sequences of a program. In this dissertation, atomicity is enforced by mining mini-graphs from basic blocks. Mining mini-graphs from larger atomic code sequences, either statically (*e.g.*, predicated hyperblocks or transactional code sequences within a software rePLay framework [71]) or dynamically (*e.g.*, from rePLay frames [78]) is possible but is outside the scope of this dissertation

Beyond the fundamental constraint of atomicity, mini-graph criteria are largely a set of tradeoffs—finding the sweet spot between the benefit of coverage and a cost, such as ease of implementation. Mini-graphs are conservatively defined so as to minimize the number of pipeline stages that are explicitly mini-graph aware, changes to the ISA, and the involvement of the operating system.

This section explores the implementation costs and coverage benefits associated

with mini-graph criteria such as mini-graph instruction formats, maximum cycle length, and which instruction types to include. The stricter the mini-graph criteria, the fewer architectural changes are required to process mini-graphs, but the less coverage these mini-graphs offer. Conversely, relaxing mini-graph constraints improves coverage but also increases the cost of implementation.

Handle formats. Mini-graph handles can be represented in a program binary using a common 32-bit instruction format—shared by Alpha, MIPS, and SPARC instructions—that assumes opcodes require 6 bits and register specifiers require 5 bits each. The remaining n bits belong to the `MGID` which can express 2^n mini-graph templates. Three possible 32-bit formats are shown in Table 2.1. With 32 bits total, a mini-graph can have either 2 register inputs and 1 register output, 3 register inputs and 1 register output, or 2 register inputs and 2 register outputs. The choice of formats is the first of many decisions that weighs coverage benefits with implementation cost.

	31 — 26	25 — 21	20 — 16	15 — 11	10 — 6	5 — 0	templates
A	opcode	input 1	input 2	output	`MGID`		2048
B	opcode	input 1	input 2	input 3	output	`MGID`	64
C	opcode	input 1	input 2	output 1	output 2	`MGID`	64

Table 2.1: Possible 32-bit Mini-Graph Instruction Formats. The size of the `MGID` determines how many mini-graph templates are supported by a single, reserved mini-graph opcode.

Although representing a handle in a standard 32-bit instruction format is the most intuitive approach—and certainly the approach most in keeping with the "spirit" of mini-graph processing—mini-graph handles are not *fundamentally* required to be encoded in a program binary with a single instruction. (At least one other form of instruction fusion, CCA graphs [23], are represented using multiple instructions.) As long they can be coalesced into a single instruction representation once inside the pipeline, an n-instruction representation of a handle is theoretically feasible.

The benefit of using more than 1 instruction to represent a mini-graph handle

is that more register specifiers and consequently more register input/output combinations become expressible. If each 32-bit instruction has room for three register specifiers (as seen in Format `A`), mini-graphs with, say 3 or 4 register inputs and 1 or 2 register outputs, could easily be encoded.

The problem with representing a handle using more than one instruction is that it sacrifices amplification of both instruction cache capacity and fetch bandwidth. A two-instruction mini-graph represented by a two-instruction handle in the instruction cache has removed no instructions at all (prior to decode) and consequently has *no* amplified instruction cache capacity or fetch bandwidth. A three-instruction mini-graph represented by two-instruction handle in the instruction cache has *half* the coverage of one represented by a single instruction handle.

Table 2.2 compares the actual coverage achieved with 1- and 2-instruction handles across the 78 benchmarks introduced in the methodology discussion of Section 1.4. Although formats `D-F`achieve more coverage than formats `A-C`, any 2-instruction handle loses half to one-third of its fetch amplification. Although all other amplification benefits remain unaffected, reduced fetch bandwidth amplification is particularly critical. Unless the processor is designed to overfetch/overdecode [90], failing to amplify fetch bandwidth effectively "pinches" an otherwise amplified pipeline. Provisioning a processor to overfetch/overdecode is contrary to the goals of mini-graph processing. Additionally, in the context of the specific goal of achieving the performance of a 4-wide processor with a 3-wide mini-graph processor, increasing coverage rates from 37% to 49% is not significant enough attempt to further simplify the processor to, say, a 2-wide mini-graph processor. With this in mind, the mini-graphs used in this dissertation use 1-instruction handles.

Register inputs and outputs. As shown in Table 2.1, there are a few ways to divide the fields of a 32-bit instruction. With 1 opcode and 3 registers (format `A`), there are 11 remaining bits for the `MGID`. This means that a single dedicated mini-graph opcode can express 2048 mini-graph templates. With 1 opcode and 4

Format	handle size	interface registers	coverage
A	1	2-in, 1-out	32.3
B	1	3-in, 1-out	36.9
C	1	2-in, 2-out	42.2
D	2	4-in, 1-out	38.0
E	2	3-in, 2-out	48.1
F	2	4-in, 2-out	48.8

Table 2.2: Coverage Achieved with 1- and 2-Instruction Handles. These coverage rates assume 7-cycle integer-memory mini-graphs.

registers (formats `B` or `C`), this leaves 6 remaining bits for the `MGID` which can express 64 mini-graph templates. The 4 register specifiers could be used for 3 register inputs and 1 register output (format `B`) or 2 register inputs and 2 register outputs (format `C`). With formats `B` or `C`, a mini-graph processor can support 64 more templates for each free opcode it can dedicate as a mini-graph opcode. For example, 4 reserved opcodes would support 256 templates. The choice of format weighs the difficulty of implementing these instruction formats, the availability of free opcodes, and the coverage gained by both more templates and a more relaxed definition of a mini-graph.

Table 2.3 shows the coverage potential associated with the three aforementioned instruction formats. (Coverage is shown for integer-memory mini-graphs supported by an ALU Pipeline of length 3.) Format `A` is the most conservative format, and offers 32% coverage. Format `B` allows an additional register input. This constraint relaxation translates to coverage of about 37%. Format `C` allows an additional output. This is the least constrained of all formats, enabling up to 42% coverage. Which format to use depends on the whether the increased coverage of more relaxed formats outweighs the cost of implementation.

Because Alpha instructions have two register inputs, Format `A` is already supported. Formats `B` and `C`, however require support for an additional register input and register output, respectively. Increasing the number of register inputs or register

Format	interface registers	No. templates per mg opcode	Coverage achieved with t templates					
			64	128	256	512	1024	2048
A	2-in, 1-out	2048	27.8	30.2	31.6	32.1	32.3	32.3
B	3-in, 1-out	64	31.3	34.0	35.9	36.6	36.8	36.9
C	2-in, 2-out	64	34.7	38.3	40.8	41.8	42.1	42.2

Table 2.3: Coverage Achieved with Possible 32-bit Mini-Graph Instruction Formats. For Formats B and C, 64, 128, 256, 512, 1024, and 2048 templates require 1, 2, 4, 8, 16, and 32 reserved mini-graph opcodes, respectively. Coverage for Format A for fewer than 2048 templates is shown in the event that a smaller MGT is used and fewer mini-graphs are selected accordingly. These coverage rates assume 7-cycle integer-memory mini-graph as defined later in this section.

outputs of an instruction requires an additional register tag and match bus in the issue queue as well as one or two additional register ports (read ports for an additional register input and write ports for an additional register output). These costs are effectively less expensive because mini-graphs otherwise amplify issue queue and register file capacity and bandwidth. In other words, the structures made larger to support three register inputs are among those that can be made smaller on a mini-graph processor in the first place.

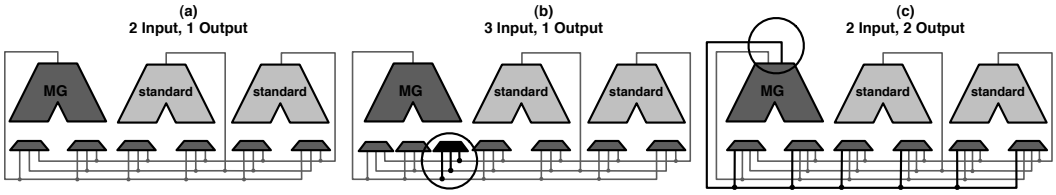


Figure 2.3: Increasing Register Inputs vs. Increasing Register Outputs

An additional register input or register output increases the bypass complexity. Figure 2.3 illustrates by considering the bypass network of 3 ALUs—1 of them supporting a mini-graph interface, 2 of them supporting an Alpha interface (see Figure 2.3a). Increasing the number of register inputs to a mini-graph from 2 to 3 requires one additional drop-point on every existing result bus, as shown in black in Figure 2.3b. An additional mini-graph register output requires an additional result bus, as

shown in black in Figure 2.3c. If 2 of the 3 ALUs supported mini-graph execution (not shown), the total area would increase further. Changes to the issue queue, register file, map table, and cross-check logic follow this general pattern; supporting an additional mini-graph output increases the area, power, and delay of structures more than supporting an additional mini-graph register input.

Allowing mini-graphs to have more than one register output would increase mini-graph coverage from 37 to 42%, but would complicate register renaming and free list management, increase the complexity of the bypass network, and require an additional register file write port for each possible in-flight mini-graph. Because the cost is greater than the cost of allowing an additional register input, format `B` represents the sweet spot of coverage and implementation ease. Mini-graphs are not unique in identifying this sweet spot: Intel, too, chose to fuse micro-ops that would collectively have three register inputs but just one register output. In the case of mini-graphs, coverage is increased from 32% to 37%, and the only implementation costs are ameliorated by the amplifying effects of mini-graphs. In order to support both 2-register-input and 3-register-input mini-graphs, formats `A` and `B` are the instruction formats used hereafter. (Technically, two formats require at least two reserved opcodes, but this dissertation will continue to refer to a generic reserved opcode `mg` as a placeholder for all mini-graph opcodes.)

Operation types. This dissertation focuses on two types of mini-graphs: *integer mini-graphs*, which contain only single-cycle ALU operations and *integer-memory mini-graphs* which can also contain, stores, loads, and conditional branches. A survey of the instructions used across the 78 benchmarks used in this dissertation shows that approximately 55% of program instructions are eligible to participate in integer mini-graphs.

Integer-memory mini-graphs expand the definition of mini-graphs to include operations beyond single-cycle ALU operations, namely loads, stores, and control instructions. (This discussion assumes that loads complete in three cycles, and stores and

control instructions in 1 cycle.) This expanded definition enables many more instructions to participate in mini-graphs. Control instructions are limited to conditional jumps, which are the most common. Unconditional jumps are sometimes eliminated from the instruction stream at decode; in such cases, placing these jumps in mini-graphs does not amplify issue bandwidth because they already consume none. Long latency operations (*e.g.*, floating point operations and and divides) would greatly increase the size of the MGT. Furthermore, amplifying execution bandwidth of floating point operations would not be as simple as for integer operations. Finally, floating point operations typically use a separate scheduler, and this dissertation assumes a unified scheduler for all mini-graph constituents. The mini-graphs of this dissertation focus on integer operations; the scope of changes are restricted to the associated integer instruction formats, scheduler, function units, and register file.

The coverage increase associated with integer-memory mini-graphs comes at an implementation cost. An integer-memory mini-graph processor makes forward reservations of multiple functional units for a single mini-graph and coordinates the forwarding and latching of transient values between these functional units; additionally, the MGT drives execution on multiple functional units. Longer execution latency requires more MGT banks.

Memory instruction limitation. Integer-memory mini-graphs may contain a maximum of one memory operation per mini-graph. Many aspects of RISC processing—memory disambiguation, load scheduling, ordering violation detection, load/store queue resource allocation—assume that memory operations can be handled individually. Not breaking that assumption minimizes changes to the micro-architecture, including such complex and latency-sensitive pieces as the load and store queues.

The one memory operation per mini-graph restriction — a practical limitation — preserves the ability to handle memory operations at an instruction granularity. As a consequence, common memory exceptions such as a TLB miss, a page fault, *etc.*

size of ALU Pipeline	integer mini-graphs	integer-memory mini-graphs
2 instruction	10.5%	33.8%
3 instruction	12.9%	36.9%
4 instruction	14.0%	38.1%
5 instruction	14.3%	38.8%

Table 2.4: Coverage Rates for 3-register-input, 1-register-output mini-graphs (as percentage of dynamic instructions) across 78 programs. Shown for integer and integer-memory mini-graphs with ALU Pipelines of sizes 2, 3, 4, and 5 instructions each. **average** (min, max)

can be associated with the mini-graph handle. Exception information is attached to the handle, the entire mini-graph is flushed, the exception is handled, and the entire mini-graph is replayed.

Execution latency. When calculating coverage, a mini-graph's length is measured by the number of its constituents; when calculating implementation cost, however, a mini-graph's length is measured by its execution latency. The execution latency determines how far in advance the scheduler is required to reserve resources such as writeback ports (see Section 2.4). Cycle count also determines the number of columns in the MGT (see Section 2.3.3), one bank per cycle. For both integer and integer-memory mini-graphs, the determining factor in execution latency is the number of ALU operations supported in a mini-graph. This also determines the required length of the ALU Pipelines (see Section 2.3.1).

Table 2.4 shows the maximal coverage rates for integer and integer-memory mini-graphs with ALU Pipelines of sizes 2, 3, 4, and 5 instructions each. The majority of mini-graphs are of size 2. Coverage increases when longer mini-graphs are supported, but this tends to taper off at 4 instructions.

Empirically, the integer mini-graphs of size 4 offer the best coverage (14.0%). Adding a fifth MGT bank to support 5-instruction integer mini-graphs is not worth the minimal (.3%) coverage increase. (Longer mini-graphs offer little increased coverage because they are still limited to just three register inputs, one register output,

and only integer operations.) Consequently, integer mini-graphs execute on 4-long ALU Pipelines, and total execution latency is no more than 4 cycles.

The "knee" of the coverage curve for integer-memory mini-graphs is at the 3-ALU or 4-ALU instruction limit. With loads that can take up to 3 cycles plus single-cycle control instructions, the maximum execution latency is 7 or 8 cycles, depending on the size of the ALU Pipeline. Because this directly impacts the size of the MGT — cycles translate to banks — and because the fourth ALU operation plays a less significant coverage role for integer-memory mini-graphs, an empirically good limitation is to have ALU Pipelines that are only 3 ALUs long.

To simplify the execution of integer-memory mini-graphs across multiple functional units, memory instructions may only occur before or after constituents that execute on the ALU Pipeline. Stores are moved past ALU instructions because they produce no transient values on which other constituents might be dependent. Atomicity restricts control instructions to the terminal position in the mini-graph. The entire sequence may have no more than one memory operation. The maximum execution latency is simply the sum of the components: `ALU-Pipe-length` + `load-latency` + `branch-latency`, in this case, 7 cycles.

Figure 2.4a shows an exhaustive list of all possible 7-cycle mini-graph patterns. The mini-graphs are ordered according to coverage; the rightmost columns show both the mini-graph's contribution to coverage as well as a running cumulative coverage score. The shaded coverage fields at the bottom of the list show those that contribute the least to coverage. For example, mini-graphs following the familiar `load-op` pattern of micro-op fusion are ranked ninth in their coverage contribution, offering only an absolute 1.74% of the total 36.94% coverage. The familiar `op-branch` pattern of macro fusion is ranked fifth, contributing to an absolute 2.33% coverage. Extending fusion benefits beyond these two idioms does, in fact, allow mini-graphs to have increased amplification benefits over more constrained forms of fusion.

These empirical data suggest simplifying the MGT and its banks. For example,

(a) Prevalence of mini-graphs in 7 MGT Banks

	1	2	3	4	5	6	7	% cov.	cuml. cov.
	int1	int2	BR					3.66	3.66
	int1	int2						3.23	6.89
	int1	int2	int3					2.81	9.70
	int1	st						2.66	12.36
	int1	int2	int3	BR				2.60	14.96
+	int1	BR						2.33	17.29
	int1	int2	int3	ld	-	-		2.16	19.45
	int1	int2	ld	-	-			1.88	21.33
*	ld	-	-	int1				1.74	23.07
	int1	int2	st					1.64	24.71
	ld	-	-	int1	int2			1.48	26.19
	ld	-	-	int1	BR			1.46	27.65
	int1	int2	st	BR				1.13	28.78
	int1	ld	-	-				1.08	29.86
	ld	-	-	int1	int2	int3	BR	**1.06**	**30.92**
	int1	int2	int3	ld	-	-	BR	**1.01**	**31.93**
	int1	int2	int3	st				**0.97**	**32.90**
	int1	int2	ld	-	-	BR		**0.86**	**33.76**
	ld	-	-	int1	int2	BR		**0.86**	**34.62**
	int1	st	BR					**0.58**	**35.20**
	ld	-	-	BR				**0.55**	**35.75**
	ld	-	-	int1	int2	int3		**0.54**	**36.29**
	int1	int2	int3	st	BR			**0.32**	**36.61**
	int1	ld	-	-	BR			**0.23**	**36.84**
	st	BR						**0.10**	**36.94**

total coverage

(b) Prevalence of mini-graphs in 5 MGT Banks

	1	2	3	4	5	% cov.	cuml. cov.
	int1	int2	BR			3.84	3.84
	int1	int2	int3	BR		3.73	7.57
	int1	int2				3.23	10.80
	int1	int2	int3			3.17	13.97
	int1	int2	int3	ld	-	2.79	16.76
	int1	st				2.70	19.46
+	int1	BR				2.42	21.88
	int1	int2	st			2.14	24.02
	int1	int2	ld	-	-	2.07	26.09
	ld	-	-	int1	BR	1.98	28.07
	ld	-	-	int1	int2	1.87	29.94
*	ld	-	-	int1		1.85	31.79
	int1	int2	st	BR		1.31	33.10
	int1	ld	-	-		1.09	34.19
	int1	st	BR			**0.56**	**34.75**
	ld	-	-	BR		**0.55**	**35.30**
	int1	ld	-	-	BR	**0.28**	**35.58**

* load-op (micro-op fusion)
\+ op-branch (macro fusion)

Figure 2.4: 7- and 5-Bank Mini-Graph Patterns. Rightmost columns of tables show coverage contribution and cumulative coverage for each mini-graph pattern. Left: Shaded percentages show patterns offering the least coverage. Dark shading indicates mini-graphs whose support requires extra logic. Right: Mini-graphs with dark shading no longer supported.

the only two patterns that utilize the 7th bank of the MGT contribute only 2.07% total. Removing the 7th bank of the MGT would provide significant area and power savings at little coverage cost. Figure 2.4b shows the result of supporting a more restricted set of mini-graph patterns. By removing the 8 least contributing patterns, two banks can be removed from the MGT. Additionally, the logic that determines which banks drive which functional units can be simplified. Notice that three less frequent patterns are still supported (see the shaded coverage fields). Although these patterns are infrequent, they require no additional hardware support (*i.e.*, not supporting them does not simplify the design) and so they are still allowed. The table on the left suggests that not supporting these mini-graphs could cut coverage by a relative 20%. In practice, however, the coverage/complexity tradeoff is good; reducing almost 30% of the MGT's real estate budget reduces coverage by less than 4%. This is because many of the unsupported mini-graphs can be broken down into smaller, supported mini-graphs. As a consequence, total coverage decreases only slightly from 36.9% in the original design to 35.6% in the simplified design. For the sake of showing maximal coverage, however, the rest of the dissertation assumes the original, 7 bank configuration.

Connectivity. Long sequences of instructions with limited number of register inputs and register outputs naturally favor mini-graphs that are connected in a dataflow graph. That mini-graphs are internally connected is likely, but not strictly required. Internally disconnected mini-graphs require no special support. Allowing disconnected mini-graphs increases the number and size of mini-graph candidates. There is, however, one potential downside; pairing two independent instructions can change their execution schedule. If this results in a critical instruction being delayed, overall slowdowns can result (see Section 3.2).

A conservative sweet spot. The set of restrictions imposed on mini-graphs represents a conservative sweet spot for maximizing coverage and minimizing support. Relaxing just one of the more easily supported restrictions would not likely

significantly increase the number of possible mini-graphs. In order to significantly increase the number of possible mini-graphs, almost all restrictions—specifically the less easily supported ones such as atomicity or a single memory operation—would need to be simultaneously relaxed. The result would be an aggregate scheme far more complicated than mini-graph processing.

Supporting the target processor. A given mini-graph processor supports only a certain number of register inputs, constituents, and instruction/cycle count. It is important that the software tool selecting mini-graphs know this configuration so that it selects mini-graphs supported by the target mini-graph processor. Incompatibilities between a mini-graph binary and the expected mini-graph processor do not result in a correctness problem. The binary simply executes less efficiently, because incompatible mini-graphs have to execute in non-mini-graph form (see Section 2.2).

2.2 Mini-Graph Encoding

Mini-graphs use an encoding scheme called *annotated outlining* that supports functional compatibility on non mini-graph processors and across different mini-graph processor implementations. Annotated outlining also enables instruction cache capacity and fetch bandwidth amplification on mini-graph processors.

Figure 2.5 illustrates the creation of an outlined executable. The original binary is shown in Figure 2.5a. First, the constituents of a particular mini-graph are *outlined*—replaced with a single control instruction that points to the new location of the outlined (as opposed to "inlined") constituents. A second control instruction following the mini-graph constituents returns the program to the outlining code's original location. Next, the constituents are *annotated*—prepended with a tag, *i.e.*, the mini-graph handle. Figure 2.5b shows the mini-graph instructions — contiguous and prepended with a handle — in outlined form. The annotated, outlined binary

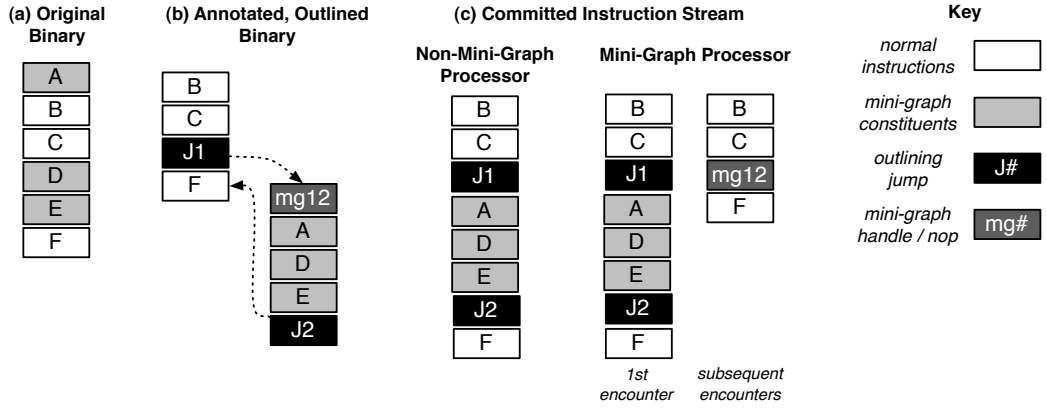


Figure 2.5: Basics of Annotated Outlining.

executes on both mini-graph and non-mini-graph processors. Figure 2.5c details which instructions are actually executed on which type of processor.

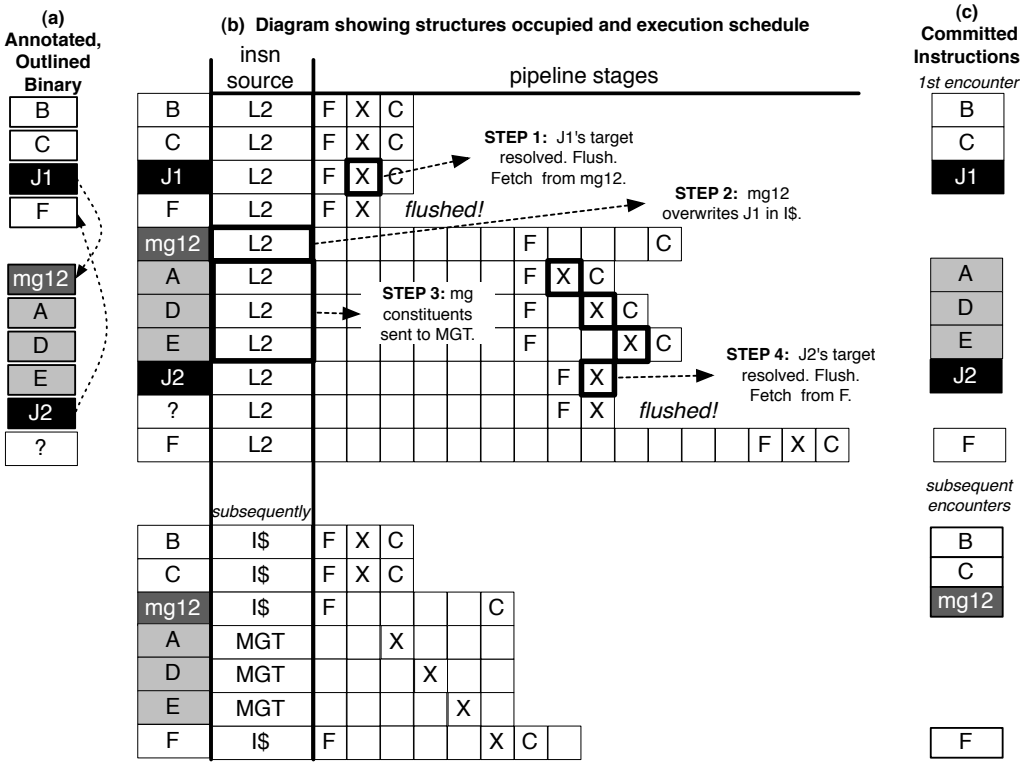


Figure 2.6: Annotated, Outlined Execution on Mini-Graph Processor.

Mini-graph execution. Execution on a mini-graph processor is shown in Figure 2.6. The "insn source" column of Figure 2.6b shows the structure from which the instructions come. Code fetched from the L2 is in its original annotated, outlined form and is placed in an instruction buffer. The mini-graph pre-processor (MGPP) sits logically between the L2 and the instruction cache, directing which instructions should be written from the instruction buffer to the instruction cache. Standard instructions and mini-graph handles are directed into the instruction cache; mini-graph constituents are pre-processed into template form and sent to the MGT.

The pipeline stages column of 2.6b shows a simplified pipeline of just fetch, execute, and commit stages. When the first outlining jump (`J1`) is resolved (step 1), the processor flushes the pipeline and begins fetching the outlined code from the L2 into the instruction buffer. The fetch unit begins fetching the annotated, outlined code (*i.e.*, the target of `J1`). Once decoded, the mini-graph handle (`mg12`) is recognized by the MGPP. It begins scanning the mini-graph constituents that follow. If the entire mini-graph is compatible with the mini-graph processor, the MGPP writes `mg12` into the instruction cache, overwriting the instruction cache entry for the most recently executed branch, `J1` (step 2). The MGPP does *not* direct the constituent instructions (`ADE`) to the instruction cache. Instead, it transforms `ADE` into their template form (`A'D'E'`) according to an algorithm explained in Section 2.5. The constituents in template form are placed in the MGT (step 3), at an index corresponding to the `MGID` in the handle. The first time a mini-graph is encountered, it executes in constituent form (see the committed instruction stream in Figure 2.6c). After the constituents are executed, the second jump (`J2`), returns control to the program main line (step 4). Execution continues, in this case beginning with instruction `F`. Subsequent encounters of the same static instructions find `mg12` in the instruction cache instead of `J1`. The processor executes the handle without encountering the outlining jumps. The MGT entry for `mg12` is accessed when the mini-graph handle reaches the execute stage.

Note that only the handle is written to the instruction cache; the other outlined instructions (`ADE`, `J2`) never enter the instruction cache. For this reason, the placement of the outlined code in the mini-graph binary has little effect on the instruction cache layout or performance. Because a mini-graph handle can be evicted from the instruction cache, the "first encounter" scenario occurs any time a handle is not in the instruction cache and needs to be fetched from the L2.

Non-mini-graph execution. Execution of an annotated, outlined binary on a non-mini-graph processor proceeds similarly to the first encounter of a mini-graph on a mini-graph processor. A non-mini-graph processor fetches all instructions from the L2 into the instruction cache. When the first outlining jump is resolved, the processor jumps to the outlined code. When the outlined code is executed, the "annotation"—the handle that precedes the mini-graph constituents—is processed as a nop, *i.e.*, ignored. The non-mini-graph processor next executes the mini-graph instructions. Finally, when the second outlining jump is resolved, the processor returns to the program main line.

The overhead is not insignificant. Each dynamic mini-graph instance requires fetching three additional instructions, including two jumps. This overhead is acceptable, however, because it occurs only in the unlikely event that a mini-graph binary is run on a non-mini-graph machine. This requires two unlikely circumstances: (1) one has access to a mini-graph binary and not the original non-mini-graph binary from which it was created and (2) one has access to a mini-graph binary and not a mini-graph processor.

A mini-graph processor can execute mini-graphs in their original, outlined constituent form. There are three contexts in which doing so is useful, each of them requiring slightly different handling. First, the processor can disable all mini-graphs across the entire program. This could aid in debugging, for example, by making execution more transparent. Non-mini-graph execution mode is signalled by setting

a user-level, per-process (non-privileged) control bit. When active, the MGPP simply treats each mini-graph handle as a `nop`, just like the non-mini-graph processor. This control bit is saved and restored as part of the processor state by the operating system; this is the *only* mini-graph processing support required by operating system.

The next two cases in which constituent execution is preferred over non-mini-graph execution is for a specific static mini-graph. This can be accomplished either permanently or temporarily. Permanent disabling of a particular mini-graph is used when a mini-graph is rejected due to correctness or performance. If a mini-graph cannot be correctly executed on a particular mini-graph processor—perhaps because it contains an unsupported constituent or its execution latency is longer than the MGT can hold—it is disabled for correctness. If a mini-graph is shown to degrade performance (see Chapter 3), it is disabled for performance. A mini-graph processor can permanently disable a mini-graph by maintaining (or restoring) the outlined form of the instructions in the instruction cache and overwriting the mini-graph handle in the instruction cache with an actual `nop`. The processor thus removes any indication that the instruction sequence was ever intended to be in a mini-graph. This process is simply repeated if the `nop` ever leaves the instruction cache.

Finally, a mini-graph processor can temporarily disable a specific mini-graph should an exception occur. Exceptions within a mini-graph may be more easily handled in outlined form, exposing the offending constituent instruction for exception handling at a finer granularity. Temporary disabling of a mini-graph is accomplished by simply re-fetching the PC of the handle into the instruction cache from the L2. This fetches the original outlining jump and constituents which are processed in non-mini-graph form just once before the MGPP kicks in and restores the mini-graph handle to the instruction cache.

Branch offsets. During the outlining process, the offsets of all PC-relative branches are updated. Branches not participating in mini-graphs are updated to reflect the code motion induced by outlining. By compressing the main line code,

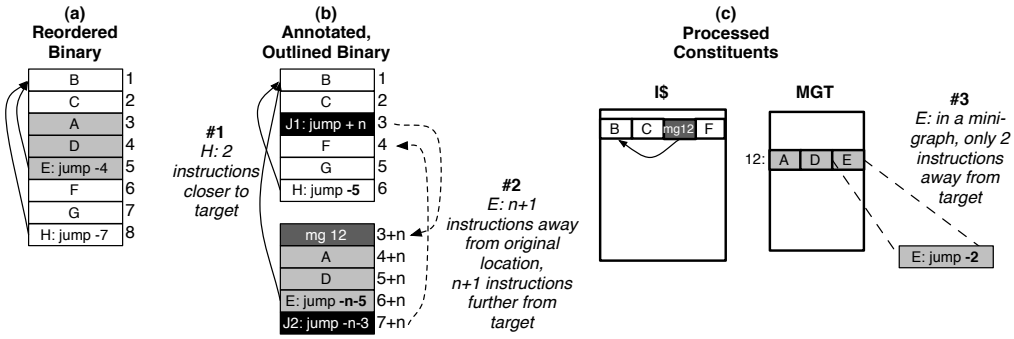


Figure 2.7: Correcting branch offsets.

outlining brings non-mini-graph branches closer to one another. All updated offsets will continue to fit in the immediate field of the branch because they are made only smaller by outlining.

Branches inside mini-graphs are updated twice: once in the binary to reflect their new outlined location in the code and then again in the MGT to reflect their "re-inlined" location once the handle has replaced the outlining jump in the instruction cache. These three cases are shown in Figure 2.7. The first case concerns control instructions not in mini-graphs. When instructions are removed from the program main line, the remaining instructions are relatively closer together. In Figure 2.7a, branch ″ targets instruction `B` with an offset of `-7`. Once mini-graph `ADE`is annotated and outlined (Figure 2.7b), its target is only 5 instructions away.

The offset of a relative branch inside a mini-graph needs to be updated to account for its new static location. In Figure 2.7a, branch `E` targets instruction `B` with an offset of `-4`. If the outlined code is n instructions from the first outlining jump, as shown in Figure 2.7b, the branch offset needs to be adjusted by $n+1$; the $+1$ accounts for the inserted handle, `mg`12. It is important that the new offset be small enough to fit in the number of bits allocated for the offset field. If n gets too large, a closer location for the outlining needs to be found.

Finally, the MGPP (discussed in Section 2.5) essentially reverses this process

when it programs the MGT. In this example, instruction `E` effectively has the PC of its handle, `mg12` which is only 2 instructions away from the target `B`. The offset of `E`, written to the MGT reflects this, as shown in Figure 2.7c.

Mini-graphs and branch prediction. When an outlining jump is first encountered and resolved it cannot be distinguished from other branches. Consequently, even though the jump does not persist in the instruction cache, the branch target buffer (BTB) is updated with the PC and target of the outlining jump, `J1`. After the mini-graph handle overwrites this jump, the newly created BTB entry leads to a branch mis-prediction upon subsequent encounters. To avoid this, the BTB entry for `J1` is cleared at the time that the outlining jump is overwritten in the instruction cache.

If a mini-graph terminates in a branch, the handle's PC stands in for the branch's PC for the purposes of target and branch prediction. The handle's PC is in fact the PC of the original outlining jump, `J1`. A new BTB entry for the mini-graph template (at the PC of `J1`) is created. A BTB entry is never created for the second outlining jump because it should only be encountered once (unless re-fetched into the instruction cache).

Other schemes. Annotated outlining can be viewed as a composite of two previous encoding techniques, outlining [20, 28, 61, 99] and annotating [90]. Classic outlining removes the mini-graph from the main line of code, which amplifies the instruction cache capacity. Annotating prepends the aggregate with an annotation, which offers binary compatibility for non-mini-graph processors that interpret the handle as a `nop`. Annotated outlining performs both of these tasks, and as a consequence reaps the benefits of both approaches. Additionally, annotated outlining amplifies a processor's fetch bandwidth by replacing the original outlining jump with the handle itself. Furthermore, because the annotation *is* the handle, knowledge of the mini-graph's interface registers and control over placement of the mini-graph in the MGT (via `MGID` choice) can be specified prior to runtime.

There are also more explicit ISA extension schemes. One commercial effort to support application specific ISA extensions is Tensilica's synthesizable processor, Xtensa [40, 41]. Programmable Instruction Set Computers (PRISC) [83] augment the base ISA with a single new instruction which exploits hardware-programmable functional units (PFU) determined at compile-time. Like `MGID` in the mini-graph handle, the PFU instruction specifies an identifier to specify which of 2048 different PFU configurations is meant. Unlike annotated outlining, the programming information for the configurations is found in the data rather than the instruction segment of the application's object file. For this reason, a PRISC binary isn't compatible with a non-PRISC processor; annotated outlining could possibly be used by PRISC to support compatibility.

2.3 Mini-Graph Execution

This section describes all the necessary structures and modifications for executing a mini-graph. The Execution Mini-Graph Table (Execution MGT) is the structure that holds all mini-graph constituents for a particular program. In the simplest case, the Execution MGT drives the execution of integer mini-graphs on a single functional unit. This functional unit is the ALU Pipeline: a multi-cycle functional unit on which an entire integer mini-graph can be executed. The ALU Pipeline can optionally be used for executing constituents two-at-a-time. In the more advanced case, the Execution MGT drives the execution of integer-memory mini-graphs across multiple function units.

2.3.1 ALU Pipelines

An ALU Pipeline is a multi-cycle functional unit. Figure 2.8 shows an ALU Pipeline with 3 external register inputs and a single register output. This particular ALU Pipeline has three ALUs and therefore supports an execution latency of three cycles.

(This can be easily extended to the longer ALU Pipelines.) The three inputs to the ALU Pipeline are latched at each cycle and made available to each ALU in the pipeline. The outputs of each stage ALU are latched to form a pipeline. Outputs from previous cycles are the transient values that are be needed as inputs in subsequent cycles. These inputs are selected with muxes. The register output of the final ALU is the register output of the ALU Pipeline. It alone is saved to the register file and broadcast across the bypass network.

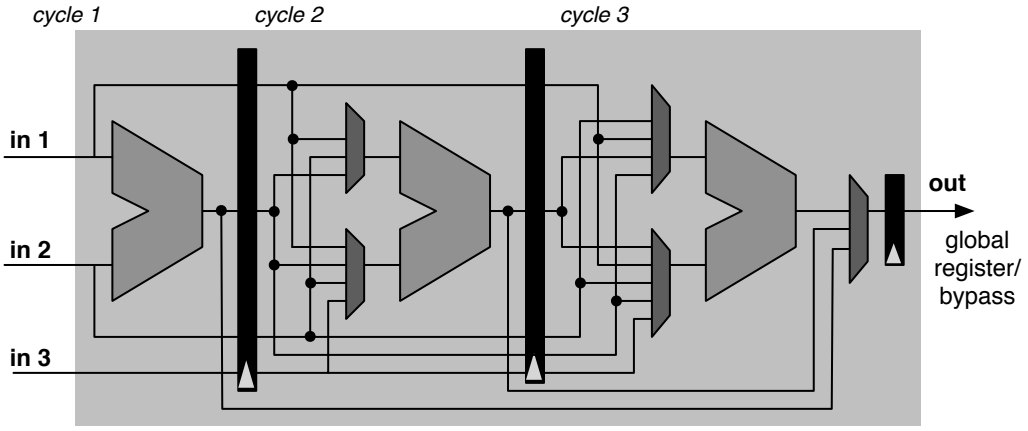


Figure 2.8: 3-Stage ALU Pipeline: 3 inputs, 1 output. Inputs are latched for ALUs to use at cycles 2 and 3.

The ALU Pipeline amplifies execution bandwidth to match the amplification of all other bandwidths that mini-graphs provide. A 3-stage ALU Pipeline can perform 3 operations per cycle, but has only 1 register/bypass output and 3 inputs, rather than 3 outputs and 6 inputs.

Early Out. The mini-graph constituent producing the mini-graph's register output might not always execute on the final ALU of the ALU Pipeline. This is because a mini-graph could be shorter than the length of the ALU Pipeline. To allow mini-graphs to broadcast their register outputs as soon as they are ready, the ALU Pipeline also has an "early out" network. The output of an ALU Pipeline is selected between the unlatched outputs of each of the stage ALUs by control signals

from the MGT.

Allowing register outputs to exit the pipeline as soon as they are ready also allows singleton instructions to execute on ALU Pipelines without penalty. A mini-graph processor can simply replace n standard ALUs with n ALU Pipelines. This keeps fixed the number of functional units that the scheduler manages as well as the complexity of the bypass network, while maintaining the number of functional units available to programs without mini-graphs.

Figure 2.9 shows the signals from the MGT. These signals are comprised of immediate and control information stored in the MGT. The control information dynamically drives the execution of each mini-graph on the ALU Pipeline. Control signals select inputs for each ALU, provide the ALU with the opcode it should perform during a given cycle, and finally determine which ALU produces the output of the ALU Pipeline for each cycle. The immediates are used as inputs to certain ALU operations.

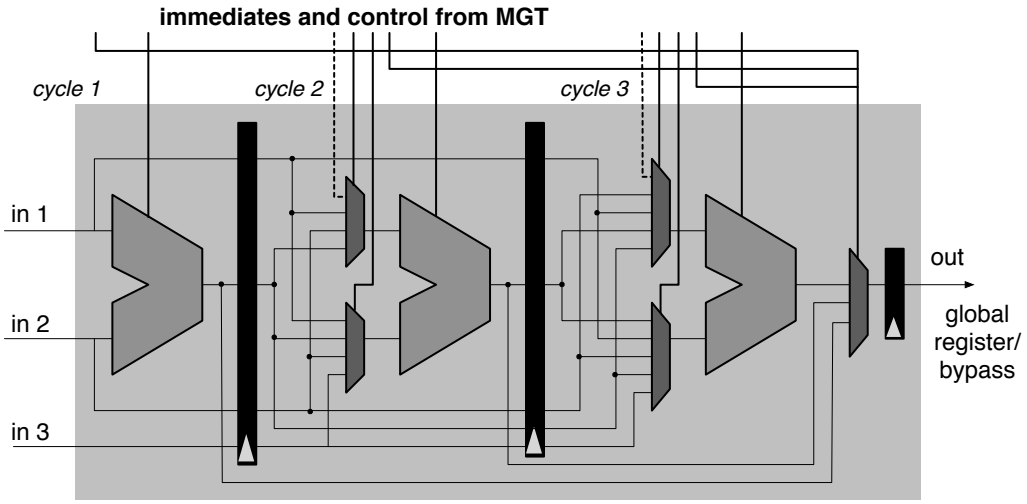


Figure 2.9: Programming the ALU Pipeline

Execution MGT with ALU Pipeline example. An example of execution on an ALU Pipeline is shown in Figure 2.10, beginning with the definitions of two mini-graphs, 14 and 20, in Figure 2.10a.

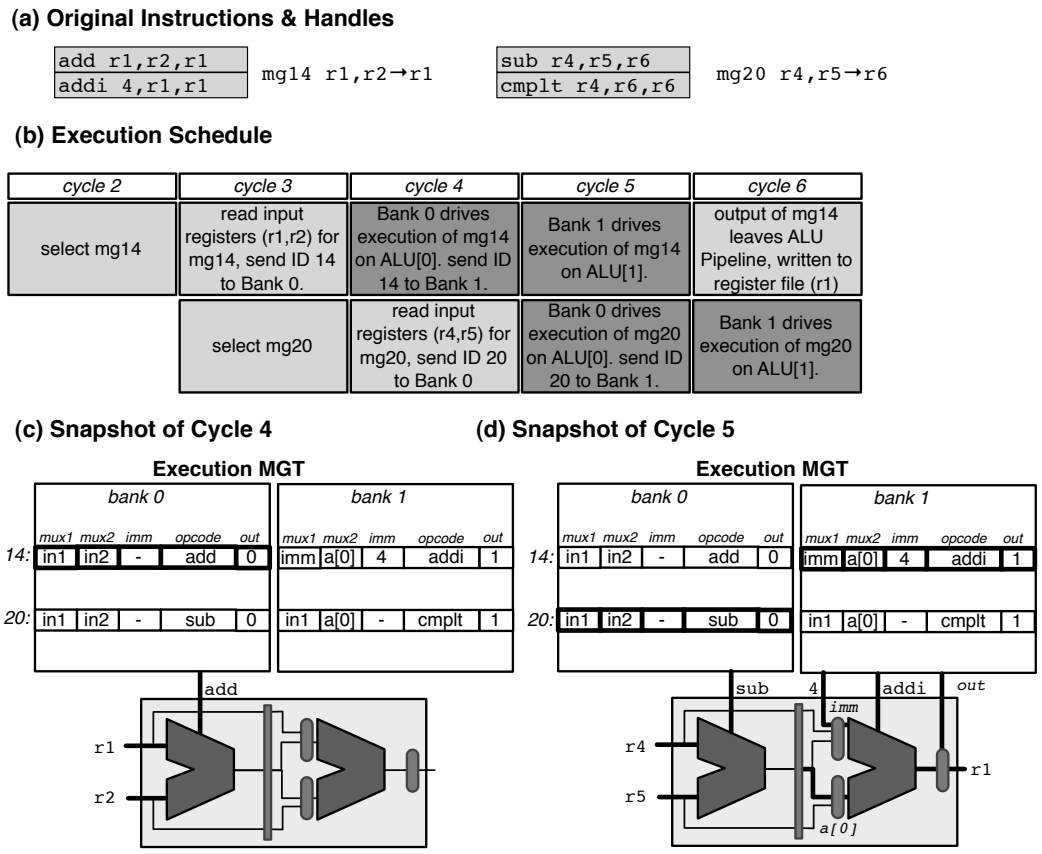


Figure 2.10: The ALU Pipeline in Action. For clarity, each cycle shows only utilized connections from Execution MGT to ALU Pipeline.

Figure 2.10b shows a detailed pipeline diagram for the scheduling, register read, and execution stages for each instruction. At cycle 2, `mg14` is selected for execution and scheduled. In cycle 3, `mg14`'s input registers (`r1`,`r2`) are read and `MGID` 12 is sent to the first bank of the Execution MGT. Concurrently, `mg20` is scheduled. In cycle 4, `mg14` is in the first stage of execution, executing the first constituent of `mg14` on ALU[0]. Concurrently, `mg20`'s input registers (`r4`,`r5`) are read and `MGID` 20 is sent to the first bank of the Execution MGT. Execution proceeds further as described in Figure 2.10b. The MGT and ALU Pipeline structures for cycles 4 and 5 are depicted in Figures 2.10c and 2.10d, respectively.

One issue that arises when multiple mini-graphs execute on the same ALU

Pipeline is the possibility of "writeback" conflicts. Because mini-graphs do not all have the same latency, it is possible that multiple mini-graphs executing on the same ALU Pipeline may produce an output at the same cycle. Unfortunately, the ALU Pipeline may only output one value per cycle. Although the MGT ultimately determines which ALU's output leaves the pipeline (notice the `out` field of the MGT), avoiding writeback conflicts is the job of the scheduler, and is described in Section 2.4.

Inter-stage connectivity and stage functionality. Each stage of the ALU Pipeline could access each of three register inputs as well as each output of the stages prior to it in the pipeline. Furthermore, each stage of the ALU Pipeline could perform various types of integer operations: arithmetic, logical, shift, *etc.*. Neither full inter-stage connectivity nor full stage functionality is strictly necessary.

The top diagram of Figure 2.11 shows a (slightly abstract) ALU pipeline fully connected with respect to inputs per pipeline stage. (Input replication for each input mux to each ALU are not shown for simplicity.) This full connectivity may be expensive to implement, particularly for the latter stages which have ever more input possibilities due to the increasing number of values produced by earlier pipeline stages. Table 2.5 shows the rates of use for each connection. Input 1 is used in `ALU[0]` (the first stage of the ALU Pipeline) by 94% of all mini-graphs, and in `ALU[1]` by 13% of all mini-graphs. The final two stages, however, use Input 1 less than 3% of the time each. This is in part due to the fact that fewer mini-graphs execute on these final stages and in part because latter constituents tend to depend on constituents prior to them.

Integer mini-graphs of length 4-instructions generate an average coverage rate of 14%. If the target ALU Pipeline does not support each input connection, fewer mini-graphs are eligible for selection. However, by removing rarely used connections, coverage is not as affected. The bottom of Figure 2.11 shows the ALU Pipeline with 5 fewer input connections; the connections with usage rates lower than 3% (shown

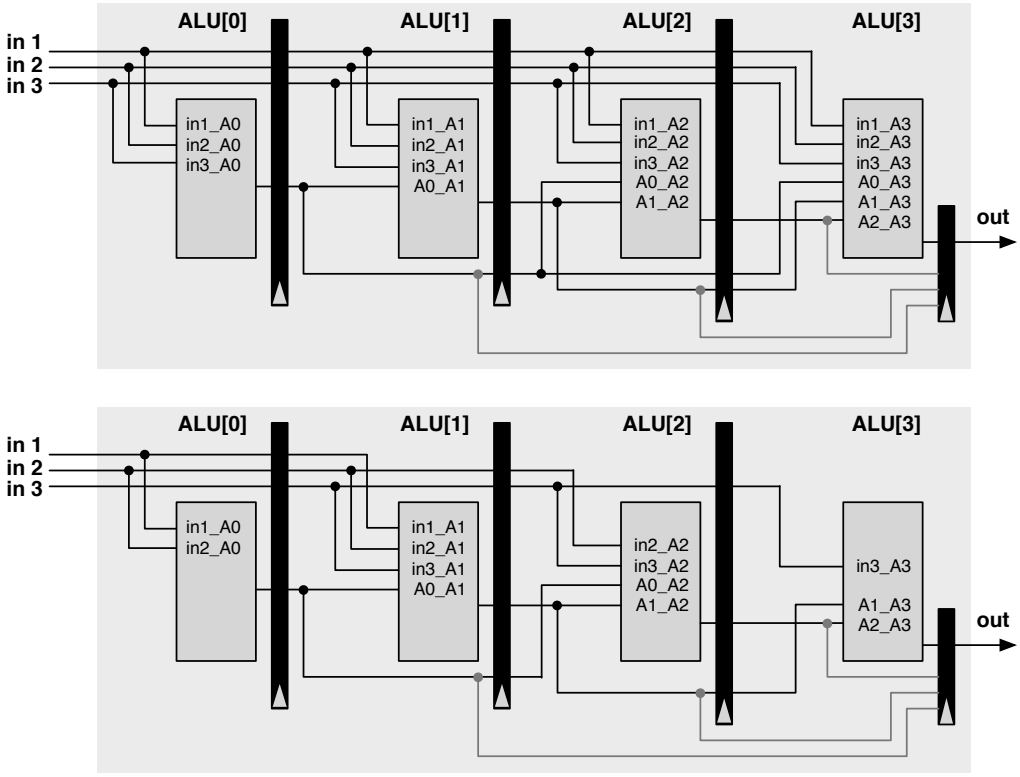


Figure 2.11: Full (top) and Reduced (bottom) Inter-Stage Connectivity within the ALU Pipeline. (top) Every pipeline stage has every possible input available to it. (bottom) Connections used for fewer than 3% of mini-graphs are no longer supported.

in bold in Table 2.5) are removed. Primarily, inputs to the final stages of the ALU Pipeline — exactly those with the most number of input sources — are pruned. The resulting coverage is still 13.5%, just down half an absolute percent from the original coverage rate of 14%.

Of the many types of integer operations (arithmetic, logical, shift, *etc.*), some are more difficult than others to complete in a single cycle. Variable shift operations, for example, can take multiple cycles in modern processors [52]. The ALU Pipeline does not need to be built to support all forms of integer operations at all stages of the pipeline. This is illustrated in a survey of 4-instruction integer mini-graphs (which run exclusively on 4-stage ALU Pipelines) as shown in Table 2.6. Arithmetic operations (spread across all 4 stages of the ALU Pipeline) account for 56% of all

Input Source	ALU[0]	ALU[1]	ALU[2]	ALU[3]
input 1	93.6 %	12.7%	**2.7%**	**1.0%**
input 2	31.7 %	37.2%	4.2%	**2.3%**
input 3	**0.1%**	16.1%	5.4%	3.7%
ALU[0]	—	93.2%	8.8%	**0.8%**
ALU[1]	—	—	26.6%	6.0%
ALU[2]	—	—	—	14.1%

Table 2.5: Input Connectivity Statistics. Average rate of how often each input connection is used across all chosen 4-instruction integer mini-graphs only. For example, 31.7% of all mini-graphs use input 2 in the first stage of the ALU Pipeline. Connections used for fewer than 3% of mini-graphs shown in bold.

instruction type	ALU[0]	ALU[1]	ALU[2]	ALU[3]
arithmetic	25.7 %	21.8%	5.5%	2.8%
logical	11.0 %	15.8%	4.9%	1.8%
shift	4.7 %	3.9%	1.6%	0.5%

Table 2.6: ALU Instruction Distribution. Rates of arithmetic, logical, and shift instructions run across each of 4 ALUs in a 4-long ALU Pipeline. Averages across 78 benchmarks for 4-instruction integer mini-graphs only.

shift operations allowed in				
all	first 3	first 2	first only	none
ALU[0]-ALU[3]	ALU[0]-ALU[2]	ALU[0]-ALU[1]	ALU[0]	none
14.0%	14.0%	13.3%	13.3%	12.0%

Table 2.7: Coverage Variation and Shift Operations. Coverage rates with varying degrees of support for shifts throughout the ALU Pipeline. Averages across 78 benchmarks for 4-instruction integer mini-graphs only.

integer operations. Logical operations account for 34% of all integer operations. Shift operations account for 11%. As shown in Table 2.6, because most mini-graphs are 2-instruction sequences, fewer operations (of any kind) occur in the final two stages of the ALU Pipeline.

It is possible to design the ALU Pipeline such that some or all stages do not support shift operations. Table 2.7 shows the coverage degradation associated with supporting shifts in fewer and fewer ALU Pipeline stages. When shift operations are prohibited from occurring in the final stage of the ALU Pipeline, coverage is unaffected. When shifts are unsupported entirely, coverage decreases from 14.0% to just 12.0%. The shifts examined here are generalized barrel shifts. Another possible compromise would be to support only small constant shifts.

Extensive exploration of reduced connectivity and stage functionality have been studied previously [21, 111] and is outside the scope of this dissertation. The remainder of this dissertation assumes full connectivity and stage functionality of the ALU Pipeline.

Superscalar ALU Pipelines. A superscalar ALU Pipeline would be possible, but would provide little benefit, because the integer components of mini-graphs as constrained have effectively no ILP. Even though mini-graphs technically do not require data dependences between constituents, mini-graphs with 3 register inputs and a single register output are most often—75% of the time—dependent chains of instructions. And in the 25% of cases where there are independent constituents, these are typically independent by virtue of a single store or branch. (Instructions

that have no register outputs tend to be more "promiscuous" in their mini-graph participation. They can be paired with any existing mini-graph without increasing the number of outputs.) These instructions are not executed on the ALU Pipeline in the first place and could not benefit from it being made superscalar. Other aggregate studies [21, 23] have needed to relax the register input and output constraints in order to exploit their proposed superscalar functional units.

2.3.2 Interlock-collapsing ALU Pipelines

A mini-graph processor's strength is its bandwidth and capacity amplification. That said, mini-graphs can also exploit execution latency reduction. Consider, for example, a four-instruction integer mini-graph. Ordinarily, each constituent requires one cycle for execution; total execution time is 4 cycles. If this mini-graph could execute on hardware that could perform two operations in a single cycle, total execution time would be 2 cycles instead. This mini-graph would not only amplify bandwidth and capacity, offering "four for the price of one," but it would also compress the height of the dataflow graph of this path by two cycles each time it is encountered.

There are many techniques for aggregate latency reduction [65, 83, 91, 112]. Although none of these techniques are fundamentally incompatible with mini-graph processing, those that use static (*i.e.*, non-reconfigurable) datapaths are a better fit for mini-graphs. The reason is as follows. In order to maximize amplification, mini-graphs support many different mini-graph templates which regularly co-occur with one another in the pipeline. A mini-graph processor regularly issues two mini-graphs per cycle. At this rate, it would be difficult to tolerate a reconfiguration penalty for each mini-graph. An alternative to supporting reconfigurable hardware is to support pair-wise stage collapsing in the ALU Pipeline. Consecutive ALU Pipeline stages are fused into a single cycle of execution. Unlike reconfigurable schemes, the ALU Pipeline's control is programmed by the MGT with no penalty.

Example. Figure 2.12 illustrates an example. Mini-graphs are shown in Figure

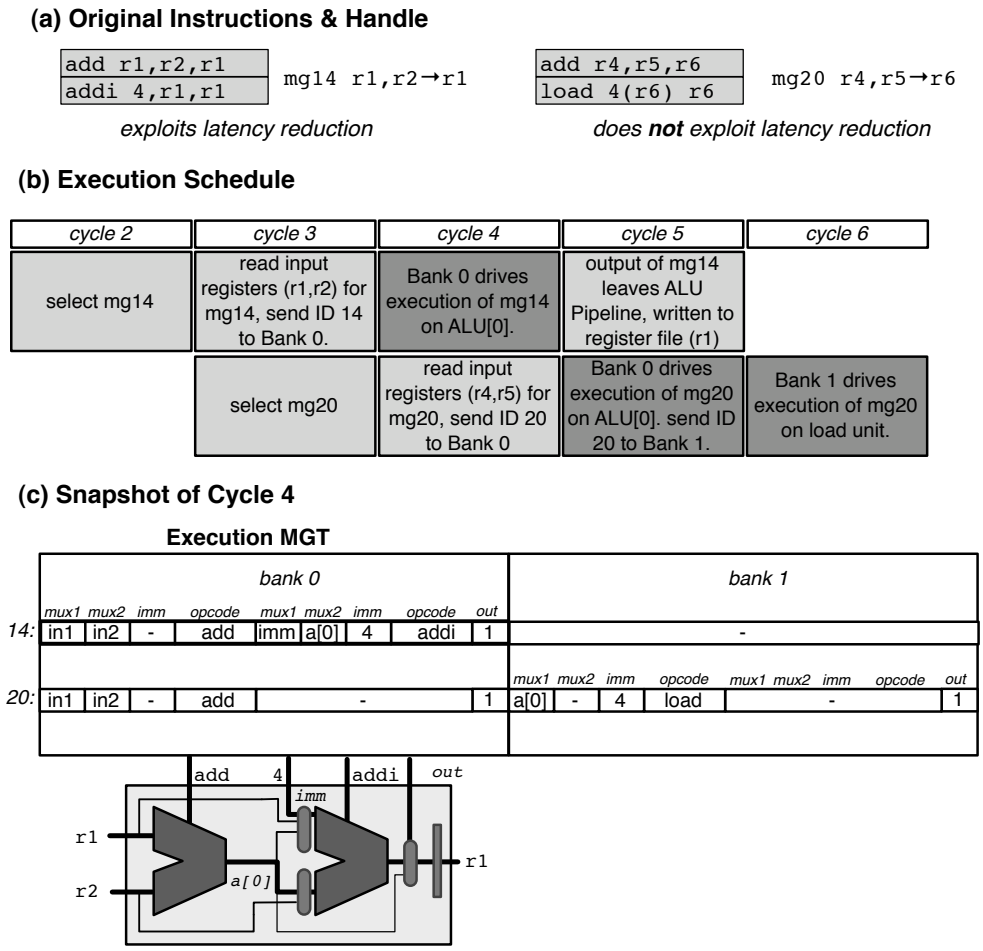


Figure 2.12: MGT support for latency reduction applied to all mini-graphs.

2.12a. Hypothetically, mini-graph `mg14`—two `add` instructions—*does* exploit latency reduction, whereas the mini-graph `mg20`—an `add-load`—does *not*. An example execution schedule is shown in Figure 2.12b.

The layout of the MGT is shown in Figure 2.12c, which depicts the programming of the ALU Pipeline at cycle 4. The MGT supports interlock-collapsing ALU Pipelines by doubling each bank and storing in a single bank the information needed to execute *two* instructions. This maintains the one-port-per-bank invariant and supports the execution of two instructions per cycle. Mini-graph entries eligible for latency reduction (*e.g.*, `mg14`) take up half as many banks as they previously did. In cases where an instruction cannot exploit latency reduction (*e.g.*, `mg20`), the second half of the bank entry remains empty. Execution of `mg20` proceeds as it did previously: one constituent per cycle and per bank.

Doubling the width of every MGT bank is expensive, but potentially justifiable, given the performance benefit of 2-to-1 collapsing over no collapsing and the prevalence of mini-graphs with 2 integer operations (recall Figure 2.4). *Tripling* the width of each MGT bank to support 3-to-1 collapsing is probably not justified, because mini-graphs with 3 integer operations are less frequent and only a subset of these are comprised of three operations that could be executed in a single cycle. Pair-wise (2-to-1) collapsing is a good cost/benefit compromise for supporting latency reduction.

ALU Pipelines can support pair-wise latency reduction in three ways. First, like CCA graphs [23] which leverage forward bypassing, the ALU Pipeline can exploit the lack of high-latency global bypass. Because there is only local/point-to-point bypass between ALU pipeline stages, values travel only short distances. A similar approach is seen in Intel's macro fusion [72, 102] which pairs a test or compare instruction with a branch on a modified branch unit that can execute both instructions in a single cycle. If leveraging forward bypassing allows two ALU instructions of any kind to execute in one cycle, then every mini-graph with two back-to-back ALU instructions

can be executed in one fewer cycles.

The impact of a latency reduction technique is measured in an alternative form of coverage. Whereas *amplification coverage* measures the number of dynamic instructions removed from a program's execution (*dynamic_instructions_removed* ÷ *total_dynamic_instructions*), *latency reduction coverage* measures the number of cycles removed from a program's execution (*dynamic_cycles_removed* ÷ *total_dynamic_instructions*). In the case of a simple pair-wise collapsing scheme, the average latency reduction coverage for integer-memory mini-graphs with up to four integer operations is approximately 13.7%. In other words, 13.7% of total dynamic program instructions experience a 1 cycle reduction in execution time.

If the lack of global bypassing is not sufficient to support pair-wise collapsing, other forms of 2-to-1 latency reduction are possible. ALU Pipelines can pair ALU operations with fast, simple logical operations and execute them in a single cycle, *a la* CCA graphs [23]. Because simple logical operations take one or two logic delays, they can be performed in the same cycle prior to arithmetic operations. ALU Pipelines can be constructed in such a way to support pairs of operations (one shorter, one longer) in a single cycle. The coverage of logical/arithmetic pairs is 5.9%. This coverage includes pairs occurring in any order.

Finally, if logical/arithmetic pairs can be executed in a single cycle, then arithmetic/arithmetic pairs, too, can be executed in a single cycle via carry-save addition, a derivative of the aforementioned logical-ALU pairing. This technique was initially proposed with interlock-collapsing ALUs [65] and has been subsequently employed by others, such as RENO [81]. By calculating partial sum and partial carry bits in parallel, carry-save addition converts two adds to an `XOR` (partial sum) and parallel `AND--OR` (partial carry) followed by an add. In other words, it can add three numbers in a single cycle. The instances of two arithmetic operations appearing back-to-back in mini-graphs is somewhat limited, offering a coverage of 4.3%.

Combining the coverage rates of both arithmetic/arithmetic and logical/arithmetic pairs yields a total latency reduction coverage rate of 10.2%, compared to the 13.7% coverage rate of a generic 2-to-1 collapsing mechanism. Given that approximately 11% of integer mini-graph constituents are shifts and 34% are logical operations (see Table 2.6) it is likely that at least some of this coverage gap could be bridged by supporting the execution of two logical operations in a single cycle. Pairs incorporating generalized shifts would be more difficult to execute in one cycle.

For all the latency reduction coverage estimates, it is important to note two things. First, coverage rates for latency reduction are much more likely to translate directly to performance improvement than mini-graph coverage for amplification. This is because the chances of latency reduction shortening the dataflow height of a program are quite high. Second, these coverage numbers assume that the selection algorithm knows nothing of any special hardware that the mini-graph processor might have. Were an ALU Pipeline to support a particular form of latency reduction, coverage would likely improve by adjusting the selection algorithm to favor the coverage of these combinations.

2.3.3 Execution on multiple functional units

Thus far, this section has described the support of integer mini-graphs that can be executed entirely on ALU Pipelines. Integer-memory mini-graphs contain instructions that execute on ALU Pipelines, as well as load, store, and branch functional units. Multiple functional units are essentially unavoidable for integer-memory mini-graphs; incorporating memory execution into an ALU Pipeline would be prohibitively complicated. Supporting integer-memory mini-graphs requires several changes to a mini-graph processor. In addition to expanding the Execution MGT to accommodate for more cycles of constituent execution, these changes include support for coordinating external and internal values of the mini-graph across functional units.

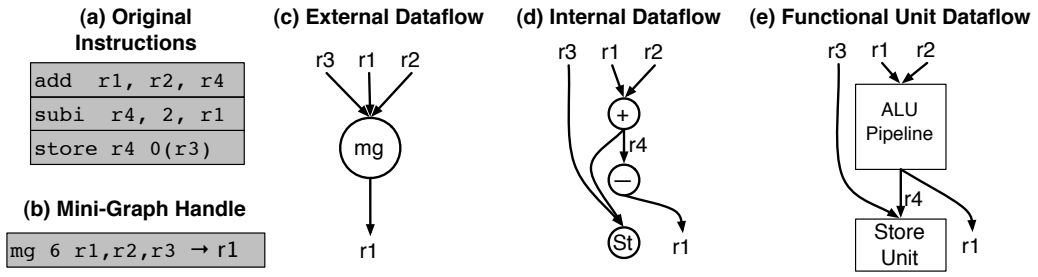


Figure 2.13: Internal and External Value Communication

External value coordination. External values are those which serve as register inputs to the mini-graph. Figure 2.13 illustrates, beginning with three constituents in Figure 2.13a and their corresponding mini-graph handle in Figure 2.13b. The external dataflow of the mini-graph is shown in Figure 2.13c, and the dataflow that occurs across the multiple functional units is shown in Figure 2.13e. When mini-graph execution occurs across functional units, not all register inputs are used at the same time. The mini-graph in this figure has three inputs (`r1`, `r2`, `r3`); only two are used by the ALU Pipeline. The third is not needed for 2 more cycles, and it is needed by the store unit.

In this example, the mini-graph's third register input can be read at the first cycle of execution of the mini-graph and latched for two cycles, or the register read can be delayed two cycles until just before the input is needed. A mini-graph processor takes the former approach. Traditionally, register (or bypass) inputs to an instruction are acquired at schedule. This invariant is maintained for a mini-graph processor's scheduler. Instead of complicating both the Scheduling MGT and the scheduler to support delayed register/bypass reads, a mini-graph processor places the responsibility with the Execution MGT. When execution begins on a non-initial functional unit, the MGT selects the correct register input from a series of latches just prior to that functional unit.

A mini-graph processor acquires all necessary inputs for a mini-graph at schedule.

Each mini-graph issues to the slot associated with the first functional unit on which the mini-graph executes; register ports and bypass inputs to this functional unit are bound to this issue slot as they are with a traditional processor. The mini-graph scheduler makes *one* register input available to all other mini-graph functional units via a single register read port exclusively dedicated to the single issue slot of the integer-memory mini-graph. This input is broadcast to the four relevant functional units. Mini-graphs requiring multiple external inputs to non-initial functional units are not supported.

Figure 2.14 illustrates. The table on the left shows the possible mini-graph configurations (originally shown in Figure 2.4) sorted according to constituents. The physical layout of the inputs to the four functional units are shown on the right at approximately the same height as the constituents on the left. All new structures and wires are outlined in dashed lines. If the functional unit is the first on which the mini-graph executes, the register input is simply the "standard input" from the register file. At the top of the figure lies the mini-graph dedicated external latched input bus from the register file. This input (coming from the register file or the global bypass) is selected by the scheduler and made available to all mini-graph functional units. Each functional unit latches this input for the number of cycles necessary to support all the mini-graphs shown on the right of the figure. Requiring each functional unit to latch the input allows there to be a single value broadcast to all four functional units per cycle. If, instead, the value were latched globally, multiple latched values would be broadcast.

The diagram for the load and store units are merged as they happen to both occur in only the first, second, third, or fourth cycle of execution. In these three cases, one, two, and three latches are required to hold the inputs until execution begins on the store unit. These latches are shown in Figure 2.14 labeled "mg-external input." Support for inputs to the ALU Pipeline and the branch unit are similar. Because mini-graph execution only begins on the ALU Pipeline at cycle one or four, the

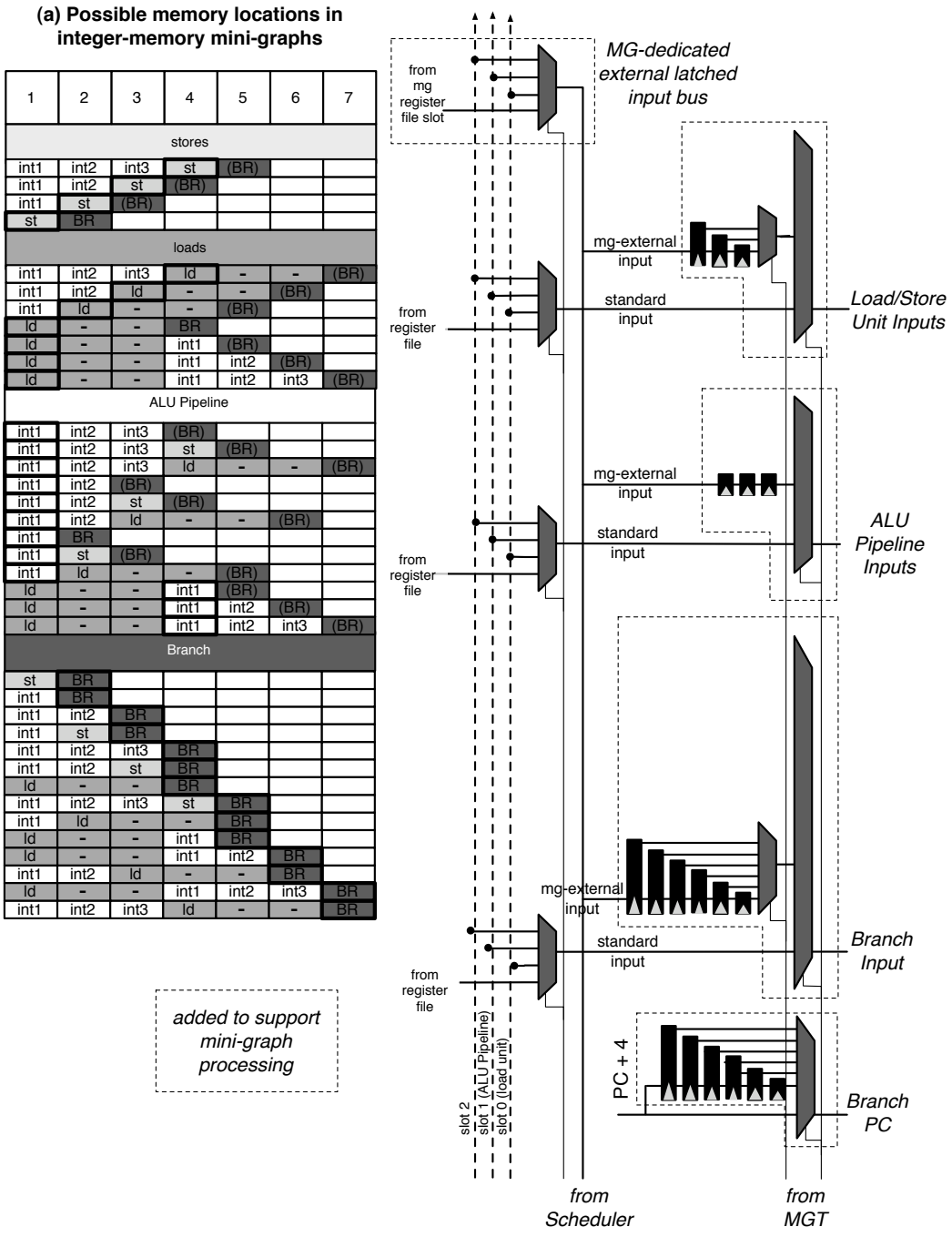


Figure 2.14: External Value Coordination.

mg-external input is only latched three times to support the latter case. Branches can occur anywhere from the second to the seventh cycle of execution and therefore require one to six latches. In addition to a register input, the branch also requires the PC+4 input. This is also latched one to six cycles. Finally, both the branch unit and both memory units may require an immediate for offsets in their address calculations. The immediate value comes either from the reservation station or the Execution MGT, depending on whether the load is a mini-graph participant or not; it does not come from the register file and requires no latching.

Internal value coordination. Transient (or mini-graph internal) values produced by one functional unit often feed constituents of the same mini-graph executing on different functional units. Because the scheduler reserves the global result bus for any value leaving a functional unit (see Section 2.4.1), these values are automatically broadcast across the existing global bypass. The scheduler cannot select this transient value from the global bypass as it normally would for bypassed values, however, because the value has no register name associated with it, nor does the scheduler know the internal register dataflow of the mini-graph. Instead, control from the Execution MGT selects this direct bypass input to the functional unit.

In some cases, the transient value is produced in a particular cycle, but not actually needed until some number of cycles later. For example, the transient value originally associated with `r4` in Figure 2.13 is produced by the first instruction of the mini-graph and broadcast immediately but the store unit does not use it until one cycle later, as shown in Figure 2.13d. The value is latched just before the functional unit and kept alive until needed.

The complete set of inputs, muxes, and latches for the various functional units is shown in Figure 2.9. The transient value enters the functional unit from the existing global bypass via a dedicated "mg-internal input" wire. In the case of the load/store units, transient values can only come from the ALU Pipeline. This value is taken off the ALU Pipeline bypass and latched every cycle. The Execution MGT

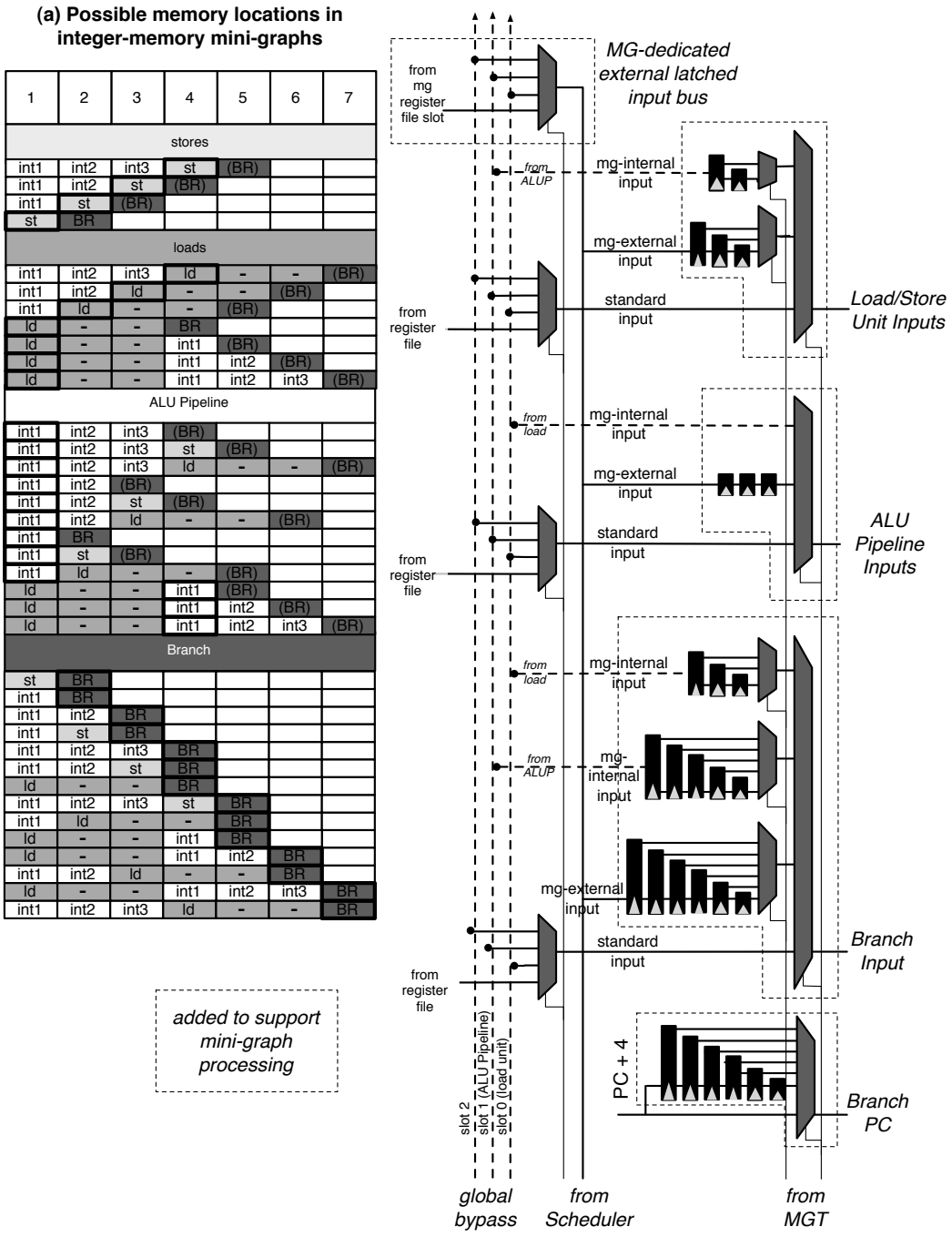


Figure 2.15: External and Internal Value Coordination.

selects both the input source and its delay. The mini-graph processor detailed in this dissertation has two ALU Pipelines, one dedicated to integer mini-graphs and one on which integer-memory mini-graphs may also execute. This distinction keeps the direct bypass from the ALU Pipeline to other functional units from being ambiguous. Even though there are two ALU Pipelines, only one of them will ever produce the value needed by the "mg-internal input" latches. The mg-internal input to the ALU Pipeline need only come from the load unit and does not need to be latched at all. The mg-internal input to the branch unit can come from either the load unit or the ALU Pipeline and is latched one to three or one to five cycles, respectively.

2.4 Mini-Graph Scheduling

The scheduler of a mini-graph processor is responsible for three basic tasks. First, it determines the latency of the register output and reserving a result bus and register write port at the appropriate cycle. Second, it reserves functional units on which the mini-graph will execute. Finally, the scheduler coordinates memory instruction scheduling, including possible load replays. These tasks apply to increasingly more specific types of mini-graphs. Reserving register ports applies to all mini-graphs. Reserving functional units in advance applies to integer-memory mini-graphs, because they execute on multiple functional units, unlike integer mini-graphs which execute on ALU Pipelines only. Finally, coordination of memory instructions applies only to mini-graphs that have memory instructions as constituents.

2.4.1 Reserving Result Busses and Register Write Ports

This dissertation explicitly couples the reservation of an issue slot with the reservation of both the result bus and and the corresponding register write port. When an instruction is issued and then executed, the result is broadcast on the result bus and concurrently sent to the register file to be written. Conventional schedulers schedule

register writes for multi-cycle operations by logically maintaining a two-dimensional reservation bitmap: one dimension represents resources (register ports), the other time (future cycles). Each cycle, the issuing instructions reserve register write ports (and result buses) by setting bits in the appropriate subsequent bitmap lines, and the bitmap advances by one line. Scheduling a three-cycle multiply reserves a write port three cycles in the future.

The primary difference between a mini-graph and a standard multi-cycle instruction (*e.g.*, a multiply) is that mini-graphs have varying execution latencies. Whereas the latency of conventional multi-cycle instructions is hard-wired into the scheduler based on opcode, the output latency of each mini-graph is calculated by the MGPP (see Section 2.5) and stored in the Scheduling MGT on a per template basis. Output latency is read from the Scheduling MGT at dispatch and stored in a field in each issue queue entry. The scheduling MGT needs one read port per mini-graph that can be dispatched in a cycle.

Coordinating transient values. An ALU Pipeline may reserve the result bus for multiple cycles: one for the register output and one for transient values sent to other functional units. An example of this is seen in the `out` field of the MGT entry for `mg20` of Figure 2.12c. The first result bus is reserved to pass a transient value from the ALU Pipeline to the load; the second result bus is reserved for the mini-graph output.

2.4.2 Reserving Functional Units

The mini-graph scheduler is a single, unified scheduler that schedules both integer and memory instructions. The scheduler represents functional unit reservations using a two-dimensional bitmap—functional unit by time. The Scheduling MGT maintains this bitmap for each mini-graph as it does the result bus/register write port information. At dispatch, the bitmap is copied from the Scheduling MGT into the issue queue.

In addition to the per-mini-graph reservation bitmaps there is a global reservation bitmap with the same configuration. The bitmap is implemented as an $n-1$ stage pipeline (or shift register) where n is maximum mini-graph execution latency (in this dissertation, 7 cycles). Each latch has a number of bits that is equal to the number of different functional units on which mini-graphs can execute. The mini-graph scheduler uses the global reservation bitmap to record which units have been reserved by recently scheduled integer-memory mini-graphs. Functional units are reserved in two stages corresponding to the *first* and *subsequent* functional units of a mini-graph.

First unit scheduling is the traditional reservation of the *first* functional unit needed by a particular mini-graph or the *only* functional unit needed by a conventional instruction or integer mini-graph. This task is essentially unchanged from the original behavior of a conventional scheduler except that the scheduler additionally consults the first stage of the global reservation bitmap to check for reservations made by previous cycles. The scheduler either incorporates the bitmap into the usual bid/grant process or broadcasts the bitmap to the issue queue and suppresses bids from conflicting instructions. The scheduler thus precludes a load from being issued if an in-flight mini-graph is already scheduled for the load unit in the next cycle.

The Scheduling MGT contains the first functional unit that each mini-graph will execute on. This information is copied to the issue queue and used during *first unit* scheduling. The first unit needed by an integer-memory mini-graph determines the scheduling slot into which the mini-graph is issued. The scheduler treats a mini-graph beginning with a load as a normal load. It, too, cannot be issued if an in-flight mini-graph is already scheduled for the load unit in the next cycle.

Subsequent unit scheduling is required by integer-memory mini-graphs because they execute on multiple functional units. To support this, the scheduling MGT contains a bitmap of the subsequent functional units required by each mini-graph.

These also go through a bid/grant process that matches (ANDs) them with the global reservation pipeline/shift-register. If any mini-graphs are selected, then their bitmap is ORed into the global reservation bitmap.

Multiple mini-graphs per cycle. A mini-graph processor requires one ALU Pipeline and one MGT read port for each mini-graph scheduled per cycle. In the case of integer mini-graphs, scheduling multiple mini-graphs is simple. Each ALU Pipeline is bound to a single scheduling and writeback slot. Just like multiple ALU operations, multiple integer mini-graphs are scheduled simultaneously with no possible resource conflict. The mini-graph processor in this dissertation supports the scheduling of only a single integer-memory mini-graph per cycle. Scheduling a second integer-memory mini-graph per cycle achieves a small (approximately 2%) performance improvement limited to those benchmarks with high coverage (and presumably already high performance gains). This gain is not enough to justify the added complexity of supporting downstream simultaneous reservations of functional units nor the ambiguity introduced in the direct bypass from duplicate functional units (*i.e.*, ALU Pipelines, load units) for transient values (recall Section 2.3.3).

2.4.3 Coordinating Memory Instructions

Load replays. The MGT implicitly assumes a fixed latency for each instruction. The latency of a load is assumed to be the time associated with a cache hit. When a mini-graph load misses in the cache, there are two possible courses of action. Misses on terminal loads are handled like misses on singleton loads. No mini-graph constituent follows the load, so the scheduler holds (or replays) all waiting instructions (which may include younger handles) as usual. Misses on interior loads are more difficult. Since it is not possible to reschedule only the subset of the mini-graph that depends on the load, the entire mini-graph is replayed. The result is a small performance penalty.

Memory instruction scheduling. If a mini-graph contains a load instruction,

the PC of the handle is used as a proxy for the PC of the memory instruction for all memory disambiguation and scheduling tasks. Mini-graph handles that contain loads are scheduled according the same policy used to schedule singleton loads. Memory scheduling mechanisms like store sets [18]—which minimally synchronize loads and stores pair-wise—are PC-based and continue to work when loads and stores embedded in mini-graphs are identified by handle PCs. As on interior load misses, the entire enclosing mini-graph is (squashed and) replayed on a load mis-speculation.

A non-mini-graph scheduler broadcasts tags in order to wake up dependent instructions eligible for scheduling in the following cycle. Because stores execute in a single cycle, a non-mini-graph scheduler couples the broadcast of store tags with the scheduling of the store. If a store is a non-initial constituent in a mini-graph, this broadcast is reserved the appropriate number of cycles in the future, extending the capability of future wakeups from those already supported (*e.g.*, loads) to store instructions. There is no threat of conflicts with other stores; the future reservation of the store functional unit already guarantees that no other store instruction needs the tag bus at that future cycle.

2.5 Mini-Graph Pre-Processing

The Mini-Graph Pre-Processor (MGPP) supports the annotated, outlining mechanism described in Section 2.2. Logically, the MGPP directs "instruction traffic" from the level two cache to the instruction cache; after leaving the level two cache, instructions reside in an instruction buffer awaiting direction. Physically, the MGPP sits after the instruction decoder, processing decoded instructions. (The decoder does not need to be replicated.) Most instructions are sent directly to the instruction cache. Annotated, outlined sequences of mini-graph constituents, however, are converted into their template form and written to the Execution MGT; a summary of their resource needs is written to the Scheduling MGT.

Instruction scanning. The MGPP scans instructions in their original fetch order. Until a handle is encountered, all instructions are simply directed from the instruction buffer to the instruction cache. When a handle is encountered, the MGPP scans the entire mini-graph. The instruction buffer is large enough to hold the maximum number of cache lines across which a mini-graph template could span. (Fetching these cache lines requires no special handling because the fetch mechanism is agnostic to the presence of mini-graph templates.) Incompatible mini-graphs are sent to the instruction cache; compatible mini-graphs are compiled into templates and sent to the MGT.

```
1. Inspect Handle:
   ID = MGT index
   map register inputs to in0-in2
   note register output
2. Inspect Constituents:
   for each constituent:
      map register inputs to in0-in2 or mg[0]-mg[2]
      map register output for downstream mappings
      note whether register output = mini-graph register output
      note cumulative execution time of mini-graph (so far)
      note functional unit required to execute constituent
3. When all constituents seen:
   output producer = last register mapping to mini-graph output
   send opcode, immediates, & mapped inputs to Execution MGT
   send writeback port and functional unit needs to Scheduling MGT
```

Figure 2.16: Pseudo-Code for Creating a Mini-Graph Template

Compiling Templates. Mini-graph constituents—as they appear in the annotated, outlined portion of a program binary—are simply regular instructions. The MGPP converts these instructions into template form before they are placed in the MGT. The pseudo-code of the conversion process is shown in Figure 2.16.

Figure 2.17 shows an example of the pseudo-code of 2.16. First, the handle is inspected (see Figure 2.17a). The `MGID` field has the value 22, so the information gathered is used to update the MGT entry at index 22. The inputs and output are recorded for use when the constituents are inspected. The MGPP uses the interface as detailed by the handle to specify the internal register dataflow of the mini-graph.

The constituent instructions immediately follow the handle. Each constituent is

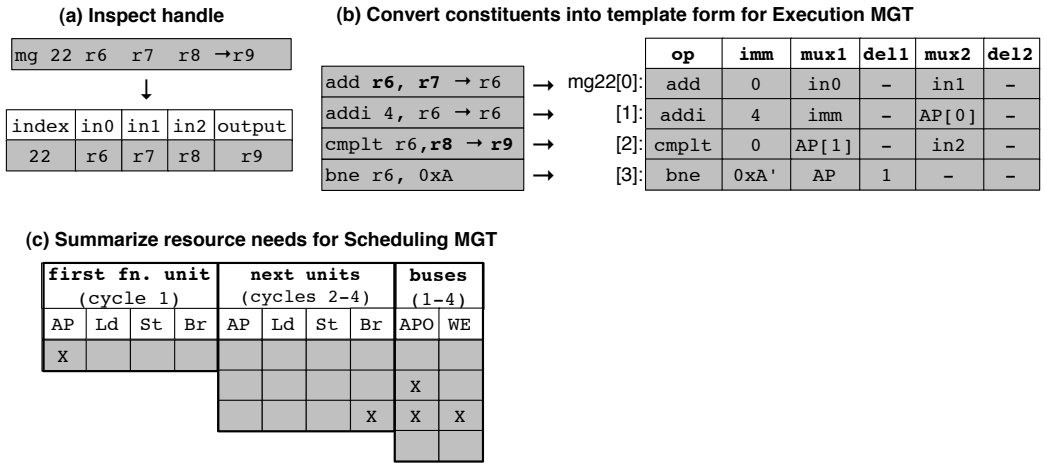


Figure 2.17: Template Compilation Example

converted to template form, as shown in Figure 2.17b. The external register interface (made bold in the figure) is specified by the handle; this informs the initial mapping of register names to input numbers. For example, `r6` is mapped to `in0`. Compiling the subsequent, internal register dataflow is similar to renaming and can be recorded on a per-constituent basis. The first instruction of the mini-graph writes to what was originally named `r6`; subsequent instructions within the mini-graph read this input as `AP[0]`—the value created by the first stage of the ALU Pipeline.

In addition to storing the necessary opcode, immediate, and mux selectors, the Execution MGT also contains control signals for input delays, `del1, del2`. These are used when inputs to the functional units require some number of latching. For example, the input to the branch uses the transient value generated by the second constituent of the mini-graph. The mux selector is set to `AP` and the delay selector is set to 1. This way, the instruction selects the output of the ALU Pipeline, latched once, as its input.

The offset of the conditional branch (shown as `0xA'`) in the fourth instruction is made relative to the PC of the handle, according to the rules specified later in this section. The completed templates with register names removed is shown in 2.17b.

In addition to creating templates used to program the Execution MGT, the MGPP also gathers the necessary information to program the Scheduling MGT, as shown in Figure 2.17c. For this, the MGPP records the functional units required by the constituents (`AP, Ld, St`, or `Br`), separating first-unit reservations from subsequent unit reservations. The Scheduling MGT reserves the ALU Pipeline at cycle 1. This takes care of the first three cycles of execution. The fourth instruction is executed on a branch functional unit, which is reserved for cycle 4.

Finally, the result busses for the ALU Pipeline is reserved. This is because ALU Pipelines have variable execution latency that depend on the instruction sequence executing on them; all other functional units have fixed latencies. In this example, a transient value created at cycle 2 that needs to be passed to the branch functional unit. The output of the mini-graph is created at cycle three. Consequently, the result bus (indicated as `APO`, for ALU Pipeline Output) is reserved at cycles 2 and 3. Only the latter also contains a write enable (`WE`) signal that indicates the value should be written to the register file. The transient value lives in the bypass network only. The write enable signal is used only by the ALU Pipeline and the load unit—the only two functional units that potentially create values needing to be written back to the register file.

2.6 Managing the Mini-Graph Table

The MGT is a cache that holds mini-graph templates and scheduling information. The MGT is indexed by the `MGID` field of any in-flight mini-graph handle. Like any other cache, the MGT can map multiple indices to the same entry. Mini-graph templates can displace one another, and MGT misses can occur during execution.

Filling the MGT. Because entries are written into the MGT infrequently (usually once per static mini-graph), these writes are not timing critical. As a consequence, a single write port is sufficient for the entire MGT; n MGT banks are written

over a period of n cycles. The MGT is a blocking cache; during these n cycles of write, dispatch is stalled. Like the penalty of executing the first instance of a mini-graph in outlined form, this stall has no performance impact over the course of an entire program run.

Handling MGT misses at dispatch. An MGT miss occurs at dispatch when the scheduling information for a particular mini-graph template is needed, but that entry has been evicted from the MGT. The mini-graph processor responds by squashing the instructions starting at the handle that triggered the miss. This instruction is invalidated from the instruction cache. When the processor re-fetches the instruction at this PC, it loads the annotated, outlined code from the level two cache, and the MGPP re-programs the MGT with the necessary scheduling and execution information. An MGT miss is expensive. In addition to the pipeline flush, the processor also incurs two back-to-back instruction cache misses in response to fetching the original outlining jump and then its target, the location of the mini-graph handle and constituents.

Avoiding MGT misses at execute. It is possible that a template present at the time of dispatch has been evicted by the time the mini-graph is ready to be executed. The processor handles the miss as it does in the dispatch case. It is possible, however, to close this "eviction window" vulnerability and keep MGT misses isolated at dispatch only. A mini-graph processor can track which mini-graphs are in flight using a counter-vector that is incremented at dispatch and decremented at commit. Using this counter, the processor can determine whether an MGT entry for a mini-graph that happens to be in-flight is about to be evicted. In such cases, the processor stalls both dispatch and the MGPP until the in-flight instruction has committed. At this point, the MGPP can evict the template and re-program the MGT with the information needed by the handle currently stalled at dispatch. When a pipeline flush occurs, the counter is also cleared to maintain precise state.

Minimizing MGT misses. There are two main ways to reduce the number of

MGT misses that occur during the execution of a program. None of these approaches are required for correctness, but they will help performance.

First, if the software tool used to select mini-graphs knows the size of the MGT, it can limit the number of mini-graph templates encoded in the binary to the number of MGT entries; each template has a dedicated MGT entry. If the program in question contains known phases, `MGID` assignments can allow non-conflicting templates to sharing MGT entries. By profiling a program and creating an interference graph between mini-graphs, static templates known not to conflict (or to rarely conflict) can be assigned the same MGT entry. This assumes prior knowledge of the MGT size and the method by which `MGID`s are mapped to MGT entries.

The second way to minimize MGT misses is in the organization of the MGT itself. When evictions and conflicts are anticipated, the MGT—like any cache—can benefit from set associativity. Specifically, the Scheduling MGT is set-associative. The Execution MGT can be direct mapped, because it is accessed after the MGT entry has already been located by the lookup in the Scheduling MGT. The Scheduling and Execution MGTs act like the tags and data of a serial tag/data cache.

A final approach to minimizing MGT misses is to disable any mini-graph whose `MGID` conflicts with an existing MGT entry. With this approach, any mini-graph with an "overflowed" `MGID` executes in outlined form. Although outlining injects two additional jumps into the instruction stream, this penalty is less than the pipeline flush followed by two instruction cache misses that occur for each MGT miss.

Virtualizing the MGT. `MGID`s map handles to templates on a per process basis. As such, the relationship between `MGID`s and the MGT is similar to that of virtual addresses and a TLB. The MGT mirrors the behavior of the TLB. If the TLB does not support process ID tags and is therefore flushed on a context switch, the MGT follows suit. When a program returns from the context switch, the MGT is re-programmed as each static handle is encountered. If, however, the TLB does support process ID's, the MGT utilizes these to support multiple processes.

The question of performance remains; inter-thread conflicts in the MGT would incur MGT misses. As suggested in the previous section, cache performance problems can be addressed by introducing a low degree of associativity. An extensive investigation of mini-graph processing in the context of multi-threaded execution is out of the scope of this dissertation.

2.7 ISA Issues

This dissertation examines mini-graph processing in the context of the Alpha ISA. Extending mini-graphs to other instruction sets is possible, but may require additional considerations. This section discusses these potential issues.

CISC instructions. The instructions executed by a Complex Instruction Set Computer (CISC) are complex instructions comprised of multiple lower-level, operations. The complex instructions are often referred to as macro-ops, the lower-level operations as micro-ops. For example, a single CISC macro-op may load a value from memory, perform some arithmetic or logical operation on the value, and then store it back into memory. In most implementations of modern CISC processors, the complex instruction is broken into its micro-op components at decode and subsequent manipulations occur at a micro-op granularity.

Processing mini-graphs in the context of CISC computing is possible on both a micro-op and macro-op level. First, any group of macro-ops that decode into a set of micro-ops that would themselves satisfy mini-graph criteria can be outlined and encoded as mini-graphs in the prescribed manner. "Mini-graphing" macro-ops applies particularly well to the many macro-ops that are actually comprised of lone micro-ops.

Second, any single macro-op that decodes into multiple micro-ops can either wholly or partially be executed as a mini-graph, assuming that it meets mini-graph criteria. In this way, mini-graph processing provides a more general, programmable

implementation of micro-op fusion; micro-ops are fused into mini-graphs after decode and processed as instruction aggregates with the aid of the MGT. "Mini-graphing" micro-ops does not require outlining. The MGT could also be hard-wired to support idioms found at the ISA rather than the program level.

Because the MGPP sits after the decoder already, forming mini-graphs after instructions have been split into micro-ops is not unreasonable. Forming mini-graphs subsequent to decode will not have the same "pinching the pipeline" effect on a CISC processor that it would have on a RISC processor; CISC instructions already compress the instruction cache footprint as well as amplify fetch bandwidth. Detailing the subsequent changes required of the encoding scheme are out of the scope of this dissertation. Fusing micro-ops into single-instruction aggregates has been the subject of other related research projects [47, 48].

Condition codes. Some ISAs support condition codes, "meta-results" corresponding to the execution of a particular instruction. If an ALU operation results in a zero or negative value, for example, a condition code can be set. Condition codes are essentially a second type of instruction output—an implicit register. Mini-graphs can be made to support condition codes, regardless of whether these codes assume single or multiple condition "registers." Supporting condition codes requires attention when creating the binary, and when renaming and executing the mini-graph.

Converting the binary of an ISA with condition codes respects the "dataflow" of the condition codes as it does for register or memory values. If a condition code dependence exists between two instructions, their relative ordering is maintained in the outlined binary when mini-graphs are formed and anchor positions are found. This added constraint may further restrict mini-graph formation. However, liveness analysis can detect which instructions actually have condition code dependences [63], making it possible to aggressively determine which instruction pairs have actual condition code dependences and which pairs do not.

Unless the condition code is both set and exclusively read inside the same mini-graph, the condition code itself is renamed. Normally, an instruction's interactions with condition codes is implicit in the opcode, but multiple mini-graphs often share the same designated mini-graph opcode. Naively assuming that all mini-graphs read and set the condition codes would induce serialization that would be prohibitive performance-wise. One option is to have multiple designated mini-graph opcodes; one that does read and set a condition code, and one that does not. Another option is to dedicate an interface register to specify condition codes. Finally, it is always possible to simply disallow condition code communication between a mini-graph and any other instruction.

The condition codes themselves should still be set at a constituent granularity. This supports intra-mini-graph (*i.e.*, transient) communication of condition codes, where one constituent reads the condition code of a previous constituent of the same mini-graph. To identify the actual constituent that sets a particular code, the processor can replay the instructions in their outlined form.

Predication. Predicated instructions are instructions whose execution is dependent upon a condition, or predicate. Only if the condition is true is the instruction executed. in *partial predication*, the predicate is simply the value of one of the existing registers. In partial predication, the predicate is already encoded in the instruction as a register input and requires no special handling. Alpha's conditional moves, for example, are naturally incorporated into the mini-graphs in this dissertation already.

In *full predication*, the predicate is either part of a special predicate register file (*e.g.*, Itanium) or is a condition code (*e.g.*, ARM). In the former case, predicate registers need to be explicitly encoded in the mini-graph handle just as standard register inputs are. Supporting condition code predicates is similar to the support for condition codes previously discussed. However, in cases where *every* instruction is predicated, the predicate reads can be made implicit. Aggregation schemes have been studied in the context of processors that support predication [22]. Finally,

predication may be beneficial as it could potentially be leveraged in a mini-graph framework to guarantee the atomicity of mini-graphs that cross basic block boundaries.

SIMD. Single Instruction, Multiple Data (SIMD) is an execution style that exploits data level parallelism. When an operation needs to be applied multiple times to many pieces of data (*e.g.*, some computation performed on each pixel of an image) a SIMD instruction can be used to apply a single instruction to a batch of inputs. There are many existing examples of SIMD instructions (Intel's MMX and SSE, ARM's NEON, SPARC's VIS, *etc.*).

This dissertation details the support of mini-graph processing in the context of the integer pipeline only. Although some SIMD implementations (Intel's SSE [49], and Freescale's Altivec [35]) support integer execution, the SIMD execution is not part of the standard integer datapath, usually leveraging a separate register file, access to memory, *etc.* So although mini-graphs would be compatible with SIMD that executes single-cycle operations in theory, in practice, no SIMD is integrated into the the integer datapath so as to make this a reality. Combining mini-graph processing and floating-point SIMD instructions would require mini-graph support to be extended to the floating point pipeline and register file accordingly. While not impossible, this extension of a mini-graph processor would be non-trivial and is beyond the scope of this dissertation.

Variable instruction length. Some ISAs encode instructions in a varying number of bytes. When instructions are fetched from the instruction cache, exact boundaries of each instruction are not known *a priori*. If a decoder can handle instructions of variable length independent of of mini-graphs, this capability should not be hindered by the presence of mini-graphs. Furthermore, a processor that supports variable instruction lengths would also support mini-graph handles of variable lengths. Variable length mini-graph handles would support a more flexible mini-graph interface. Each template could be expressed by a single unused opcode

followed by a longer, dedicated `MGID` field. More bytes would support more inputs and outputs. Of course, these capabilities only change the cost/benefit equations; pros and cons would still need to be weighed before settling on an encoding scheme in this new context.

Forward Compatibility. Mini-graph encoding relies on the existence of unused opcodes that identify mini-graphs. Once a processor has committed to supporting mini-graphs, these dedicated mini-graph opcodes cannot be used to encode new, non-mini-graph instructions or else mini-graph handles would be mis-interpreted by new processors. That said, subsequent processors are *not* required to continue to support mini-graphs; the dedicated opcodes can once again be interpreted as a nop, effectively disabling all mini-graphs and reverting to constituent execution in outlined form.

2.8 Architectural Issues

This dissertation examines mini-graphs in the context a microarchitecture similar to an unclustered Alpha 21264, specifically one that implements register renaming using a unified physical register file (PRF) (as opposed to an architectural register file and a value-based ROB), that has a unified scheduler for integer and memory operations and a separate scheduler for floating-point operations.

Register renaming style. A processor that performs renaming with an architectural register file (ARF) and a value-based ROB can both support and benefit from mini-graph processing. In the context of a PRF, instruction inputs are read from the PRF at dispatch and outputs are written to the PRF at writeback. In the context of an ARF, instruction inputs are read from the ROB or ARF at dispatch and outputs are written to the ROB a writeback and the ARF at commit. As a consequence, mini-graphs amplify both ROB and ARF read and write bandwidth. ROB and ARF read bandwidth is amplified at dispatch, ROB write bandwidth at

writeback, and ARF write bandwidth at commit. Whereas mini-graphs in the context of a PRF amplify PRF capacity, mini-graphs in the context of an ARF amplify ROB capacity.

Clustered architectures. A clustered microarchitecture can easily support mini-graph processing. The steering algorithm operates on handles at it would for any other singleton instruction. The only caveat is that the cluster to which a mini-graph is sent have the local resources necessary to execute that mini-graph. For most mini-graphs, for example, the target cluster needs access to an ALU Pipeline.

Memory models. Because mini-graphs may contain at most one memory instruction, supporting mini-graphs is completely orthogonal to the memory model maintained by the processor. Whether it support sequential, relaxed, or weak consistency, the processor need no extra effort to maintain correctness when executing a mini-graph binary. The processor simply handles any mini-graph with a load/store in it as it would any singleton instruction with a load/store in it.

2.9 Summary

There are any number of instructions that might be embedded in mini-graphs. This dissertation isolates two forms of mini-graphs. *Integer mini-graphs* are mini-graphs that contain only single-cycle ALU operations. Integer mini-graphs execute entirely on ALU Pipelines driven by control from the Execution MGT. A scheduler need only know the execution length and it can schedule any mini-graph as it would a multi-cycle operation. Integer mini-graphs require minimal support, but are fewer and smaller and therefore have less impact, both in terms of resource amplification and performance. Integer mini-graphs have coverage rates of 14% on average. However, even these mini-graphs could be used to support limited latency reduction, which could compensate for some lack of amplification with a non-trivial performance boost.

Integer-memory mini-graphs are a superset of integer mini-graphs; integer-memory mini-graphs may additionally contain memory instructions as well as control instructions. The execution of integer-memory mini-graphs occurs across multiple functional units. The integer portion of integer-memory mini-graphs are executed on ALU Pipelines. Memory and control operations are sent to existing functional units. The scheduler of a mini-graph processor that can process integer-memory mini-graphs is modified to make forward reservations of multiple functional units and result buses/register write ports. Latches are also needed to store transient values prior to their use by some functional units. For this added support, integer-memory mini-graphs have greater impact; coverage more than doubles into the 30-35% range on average.

Several restrictions follow from the desire to process mini-graphs as singleton instructions; each ensures that mini-graphs have minimal impact on the existing microarchitecture, ISA, and the operating system. First, a mini-graph has the register interface of a singleton RISC instruction (up to three inputs, one output); this allows a mini-graph to be processed on the existing mechanisms that coordinate register renaming, scheduling, bypassing, and writeback. Integer-memory mini-graphs are constrained to have at most one memory reference and one control transfer. This preserves the processor's instruction-level handling of memory and control operations, as well as memory-related exception handling. To preserve atomicity, mini-graph constituents cannot cross basic block boundaries. Control instructions are allowed in integer-memory mini-graphs, but only in the terminal position.

Mini-graphs use an encoding scheme called *annotated outlining*, which supports functional compatibility on non mini-graph processors and across different mini-graph processor implementations. Annotated outlining also enables instruction cache capacity and fetch bandwidth amplification on mini-graph processors.

Chapter 3

Mini-Graph Selection

This chapter describes and evaluates techniques for selecting mini-graphs. The chapter begins by introducing several techniques for maximizing coverage. The goal of maximizing coverage is reasonable, because the greater the coverage—the more dynamic instructions that are embedded into mini-graphs—the more impact mini-graph processing can make. A brief evaluation of the coverage-maximizing algorithm exposes a potential performance problem associated with mini-graph processing called *serialization*. Serialization is the introduction of an artificial dependence between two instructions by virtue of being placed in a mini-graph. This dependence can delay the execution of instructions found within a mini-graph, making them execute later than they normally do in singleton (*i.e.*, non-mini-graph) form.

The benefit of mini-graph processing is capacity and bandwidth amplification—a second-order performance effect, but the cost of serialization is increased latency—a first-order performance effect. As a result, serialization can degrade IPC, even to the point of overwhelming the benefits of mini-graph processing. Aggressive selection schemes that maximize amplification produce high amplification rates, but—due to serialization—cannot use this amplification to simulate a commensurate increase in physical resources without great IPC penalties. Conservative selection schemes avoid serialization by static analysis, but produce only half the amplification rates of their

aggressive counterparts. This prohibits using mini-graph processing as a replacement for superscalar width; if there aren't enough mini-graphs, the amplification does not match the effect of actually increasing processor resources.

In response to this problem, the second part of the chapter focuses on serialization-aware mini-graph selection—the exclusion of harmful mini-graphs in order to maintain robust performance. To reconcile the seemingly conflicting goals of resource amplification and serialization avoidance, this chapter develops four schemes that identify and reject mini-graphs with harmful serialization:

- $\text{Struct}_{\text{None}}$ uses program structure to accept only mini-graphs not subject to serialization.

- $\text{Struct}_{\text{Bounded}}$ uses program structure to accept mini-graphs whose delay can be qualitatively bounded.

- $\text{Slack}_{\text{Profile}}$ uses local slack profiles [32] 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.

- $\text{Slack}_{\text{Dynamic}}$ is a hardware implementation of $\text{Slack}_{\text{Profile}}$. It monitors actual execution to identify and disable mini-graphs that actually suffer from serialization delay and whose delay is actually propagated to consumers.

The most effective of these, $\text{Slack}_{\text{Profile}}$, uses local slack profiles to reject mini-graphs whose estimated delay cannot be absorbed by the rest of the program. $\text{Slack}_{\text{Profile}}$ virtually eliminates serialization-induced slowdowns while maintaining high amplification rates.

The rest of this chapter is as follows. Section 3.1 introduces the basic, coverage-maximizing mini-graph selection algorithm. Section 3.2 introduces the problem and

causes of serialization. Sections 3.3 and 3.4 presents the four serialization-aware selection algorithms. Section 3.4.3 presents a detailed analysis of all selection algorithms using an exhaustive limit study. Finally, Section 3.5 discusses the applicability of these selection policies to aggregate schemes beyond mini-graph processing.

3.1 Basic Coverage Maximizing Selection

Mini-graphs are selected off-line by a software tool (compiler or binary rewriter) that identifies instruction groups that satisfy mini-graph criteria and then encodes them into the executable. The goal of the most basic mini-graph selection algorithm is to maximize *dynamic coverage*, the percentage of original program dynamic instructions that are "embedded" in handles and eliminated from the pipeline. Higher coverage means more resource amplification.

This section describes this basic coverage-maximizing mini-graph selection algorithm, which is a simple two-step process. First, an initial pool of static mini-graph candidates are identified in the program binary. This step is described in Section 3.1.1. Second, a greedy algorithm selects non-overlapping mini-graph templates from this initial pool. This step is described in Section 3.1.2. Two improvements to the basic greedy algorithm are subsequently introduced in Sections 3.1.3 and 3.1.4. Finally, Section 3.1.5 compares the greedy approach with exhaustive methods.

3.1.1 Mini-Graph Identification

An offline software tool enumerates all legal mini-graphs in a program. The identification algorithm performs an exhaustive backward search from each instruction in the program, enumerating all possible legal mini-graphs. Enumeration is exponential in the number of instructions considered, but since mini-graphs are restricted to basic-blocks, the number of instructions under consideration at any time is typically small. This enumerated list is the pool from which the actual mini-graphs are

selected.

Although the instructions in a mini-graph are not necessarily contiguous in the original program, execution semantics do not change when they are collapsed to a single handle. Each mini-graph needs a legal *anchor* location around which to collapse the constituent instructions. Moving constituents to the anchor location can not result in load/store reordering or register dependence violations [90]. If a mini-graph contains a control transfer, this is the anchor location.

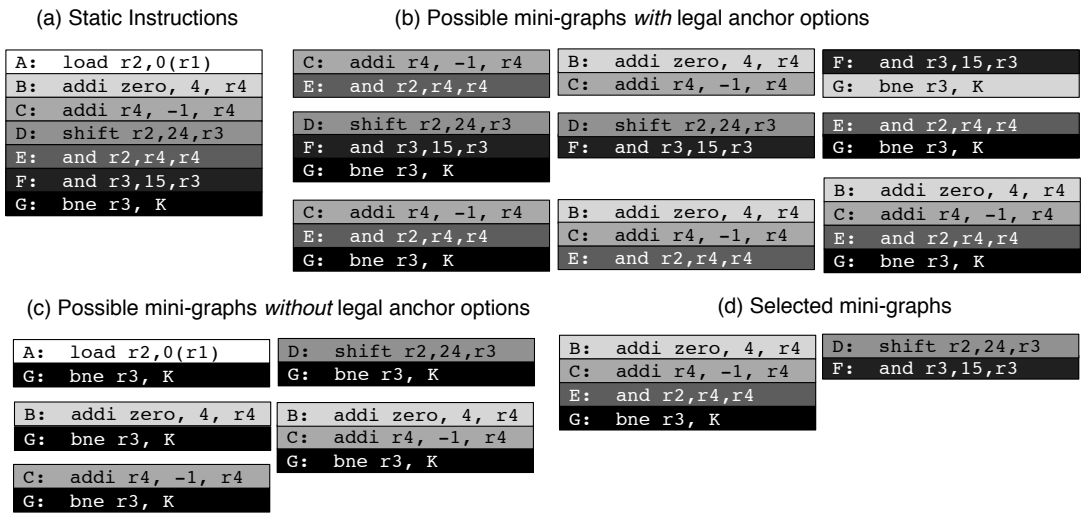


Figure 3.1: Mini-Graph Identification. (a) 7-instruction basic block from SPEC benchmark *eon*. List of mini-graphs with legal (b) and illegal (c) anchor positions. (d) Mini-graphs chosen with greedy selection algorithm.

Identification example. Figure 3.1 shows an example of mini-graph identification. Figure 3.1a shows a 7-instruction basic block from the SPEC benchmark *eon*. Of this basic block, an exhaustive search identifies 14 possible mini-graphs. Of these, 9 are "legal" in that they can be moved to a single static location that maintains control flow and creates no register dependence violations (see Figure 3.1b). (Because this basic block contains only one load, load/store reordering is not a concern.)

The other five mini-graphs (shown in Figure 3.1c) are "illegal" because they create a register dependence violation. For example, the first mini-graph of Figure 3.1c, `AG`, consists of a load and a branch. In order to maintain the original control flow,

instruction `A` (the load) should be moved past instructions `B-F`. This move violates the `r2` register dependence between instruction `A` and its consumers `D` and `E`.

It is possible that some register dependence violations occur as a result of the register name assignment and are not due to an actual dependence violation. For example, instructions `C` and `E` both write to register `r4`. As a result, moving instruction `C` below instruction `E` creates a register naming violation that simply does not occur if instruction `C` had instead been assigned a different output register, say `r5`. In such cases, changing the register assignment does solve this violation. In practice, however, this scenario is rare. In fact, the *real* reason instruction `C` cannot be moved past instruction `E` is because `E` depends on `C`: `r4` is also an input to `E`. Of the five illegal examples shown in Figure 3.1c, none of them are made legal by changing the register names. Due to the infrequency of this scenario, this optimization is not further explored. In the context of an ISA with fewer register names available, such a study might prove more worthwhile.

3.1.2 Greedy Selection Algorithm

Once all possible, legal mini-graphs have been identified, mini-graph selection begins. Among the benchmarks studied in this dissertation, mini-graph identification produces a candidate pool of anywhere from thousands to tens of thousands of possible mini-graphs per program and anywhere from tens to hundreds of possible mini-graphs per basic block. For performance reasons, not all of the mini-graphs in the candidate pool are encoded in the binary. Mini-graphs dynamically populate a fixed-size structure, the MGT. If more mini-graphs are chosen than there are entries in the MGT, mini-graphs share MGT entries, which potentially leads to MGT conflicts at runtime. An MGT miss results in a pipeline flush followed by two consecutive instruction cache misses. This event does not affect performance if it occurs one time per static mini-graph, but it does affect performance if it occurs one time per

dynamic mini-graph. One way to minimize MGT misses is to constrain the mini-graph selection algorithm to choose only as many mini-graph templates as can fit in the MGT. This requires prior knowledge of the MGT capacity. A simple, greedy algorithm selects the most beneficial mini-graphs until the MGT budget is reached.

MGT entry sharing. It is also possible to select more mini-graphs than there are MGT entries and coordinate MGT entry sharing with no performance penalty. This is done by leveraging profiling information about a program's behavior. If two mini-graphs occur within different phases of the same program, they can use the same MGT entry for their respective program phases without performance-hindering conflicts in the MGT. This assumes prior knowledge of both the MGT size and the method by which `MGID`s are mapped to MGT entries. The performance data shown throughout the dissertation assume a knowledge of the MGT capacity but do not exploit entry sharing techniques as they were found to offer limited coverage improvements in practice. (Under a constrained MGT budget, the most beneficial mini-graphs are those which occur so frequently throughout the program that sharing an MGT entry with much less frequent mini-graphs has little impact.)

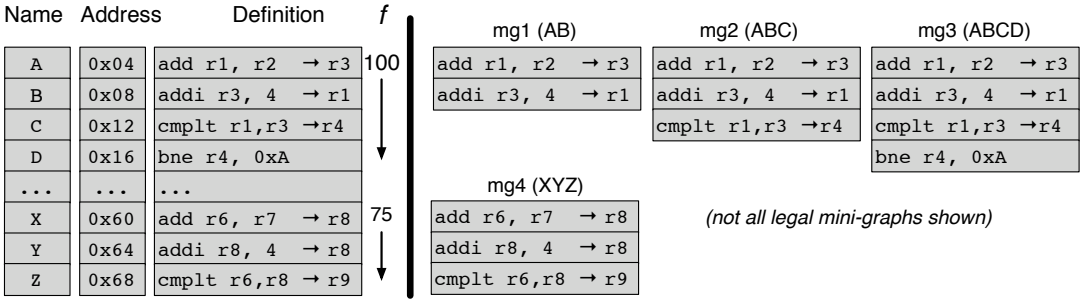


Figure 3.2: Static instructions and their mini-graphs. f indicates frequency.

A simplified starting point of the mini-graph selection algorithm is is illustrated in Figure 3.2; inspection of the static instructions `A-Z` (left) identifies four possible mini-graphs `mg`1-`mg`4 (right). The goal of the basic, greedy selection algorithm is to maximize dynamic coverage, the amplification benefit offered by each mini-graph.

The algorithm proceeds iteratively, selecting mini-graphs until the candidate pool is empty or the MGT budget is reached, whichever comes first. Each iteration has two simple steps, as shown in the pseudo-code at the top of Figure 3.3. Step 1 is to choose the mini-graph from the candidate pool with the largest coverage. Step 2 is to remove any mini-graphs with shared constituents from the selection pool.

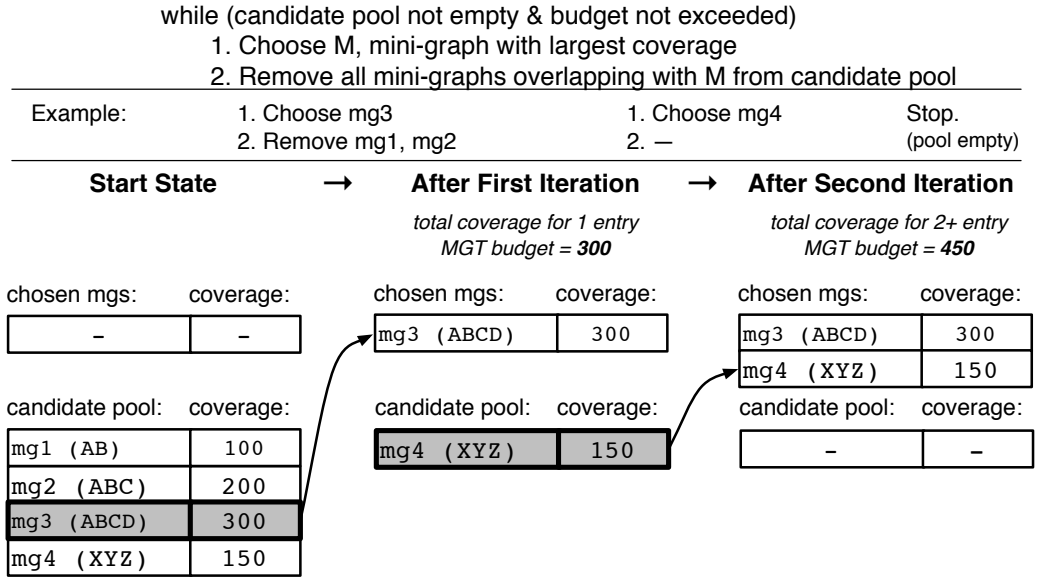


Figure 3.3: Greedy selection algorithm. Pseudo-code (top) and example (bottom).

Estimating coverage. Coverage can be approximated as $(n-1)*f$, where n is its size in instructions and f is its execution frequency. A basic-block frequency profile approximates f; frequencies of each static instruction in the current example are indicated in Figure 3.2. For example, `mg3`, with a size of 4 and a frequency of 100, has a coverage score of 300. The smaller, less frequently occurring `mg4` has a coverage score of only 150.

Removing overlapping mini-graphs. Once a mini-graph is selected, any mini-graphs with shared constituents are removed from the selection pool. Allowing a single static instruction to occur in two distinct mini-graphs duplicates the instruction from the original program. On the one hand, instruction duplication

potentially allows more instructions to participate in mini-graphs. If an instruction has two consumers and is duplicated and placed in two mini-graphs—one with each consumer—that instruction no longer requires an output register. As a consequence, another constituent that does have a register output can be included in the mini-graph. An upper bound on the increased coverage offered by replication is to allow mini-graphs to have 2 register outputs (one output is assumed to be made entirely transient due to replication). The coverage improvement is at most 5% (see Format `C` in Table 2.2 of Section 2.1). On the other hand, duplicating instructions actually *increases* the operation count of the program. Even if the execution bandwidth amplification offered by ALU Pipelines could mitigate this increased operation count, instruction duplication runs contrary to the goals of mini-graph processing and has ominous energy implications.

A working example of the greedy selection algorithm is shown at the bottom of Figure 3.3. The start state shows the initial candidate pool with each mini-graph's coverage score. In the first iteration, `mg3` is selected and the overlapping mini-graphs, `mg1` and `mg2` are removed from the selection pool. At this point, total coverage is 300. If there were only room for a single mini-graph in the MGT, this would be the final coverage score. In the second iteration, the final mini-graph, `mg4` is selected, yielding a final coverage score of 450.

3.1.3 Template Sharing

The basic, greedy algorithm presented in the previous section yields high coverage rates when the MGT budget is large. When the MGT budget is small, however, the algorithm shows room for improvement. The greedy algorithm assigns a single static mini-graph to each MGT entry. One way to maximize the utility of each MGT entry is to explicitly coordinate the sharing of the same mini-graph template across multiple static mini-graphs.

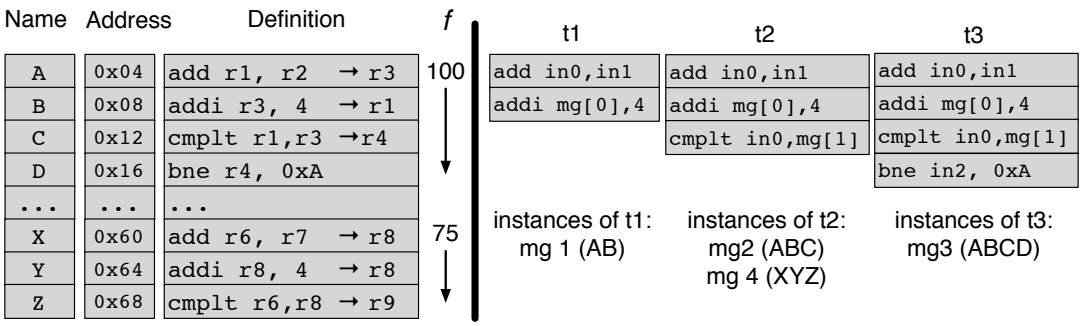


Figure 3.4: Static instructions and their mini-graph templates. f indicates frequency.

Forming templates. The template encodes the mini-graphs's internal definition. A mini-graph template specifies the exact constituent operations (opcodes and immediates) as well as the internal register dataflow of the mini-graph. Figure 3.4 illustrates. The hypothetical stream of static instructions, `A-Z`, and the four mini-graphs, `mg`1-`mg`4, are unchanged from Figure 3.2, but the mini-graph templates, `t1-t3`, are new. This is the information occupies the MGT.

Two semantically equivalent mini-graphs—`mg`2 and `mg`4, for example—may use the same mini-graph template even if they have different register names. Register names are used by the handles and act as parameters to a particular static instance of a mini-graph. Due to space constraints within the handle, mini-graph templates are not parameterized by immediates or operations; static mini-graphs using the same template agree on all opcode and immediate values within the mini-graph. A discussion of parameterizing immediates under a comparable use of templates has been discussed elsewhere [25, 26].

By incorporating the ability of multiple static mini-graph instances to use the same mini-graph template (and therefore a single MGT entry) the basic greedy algorithm can yield improved coverage under smaller MGT budgets. Pseudo-code for the modified greedy algorithm is shown at the top of Figure 3.5. After mini-graphs are formed into template groups, the algorithm proceeds almost identically, operating on templates rather than individual mini-graphs. The template with the

Form template groups.
while (**template** pool not empty & budget not exceeded)
1. Choose **T**, **template** with largest coverage
2. Choose M, all mini-graph instances of T
3. Remove all mini-graphs overlapping with instances of M from candidate pool
4. Update coverage of remaining templates in template pool

Example:
1. Choose t2
2. Choose mg2, mg4
3. Remove mg1, mg3
4. t1 & t3 now empty (*i.e.,* no coverage)

Stop.
(pool empty)

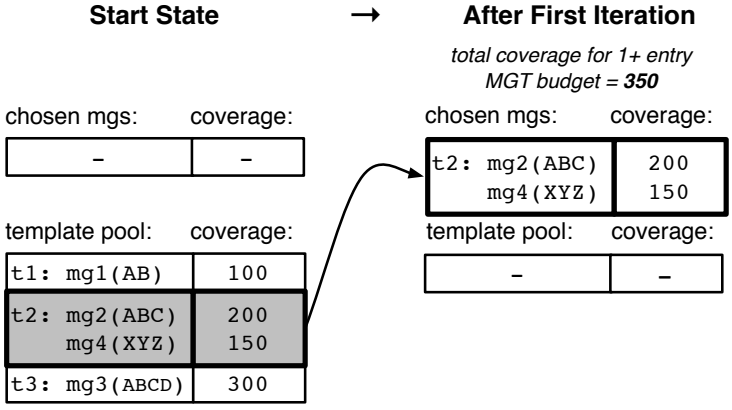


Figure 3.5: Greedy selection with template sharing. Pseudo-code (top) and example (bottom). Modifications to previous algorithm shown in bold.

highest coverage is chosen first. All mini-graph instances of this template are selected and any overlapping mini-graphs are removed from the template pool. At the end of each iteration, coverage scores of the remaining mini-graph templates are updated. The process repeats until the template pool is exhausted or the MGT budget is reached.

Estimating and updating template coverage. The coverage score of a mini-graph template is still $(n-1)*f$, where n the template size in instructions and f is its execution frequency. The execution frequency, however, is now the sum of the execution frequencies of all of a template's static instances. Once again, a basic-block frequency profile approximates f. When a mini-graph is removed from the template pool because it overlaps with a newly-chosen mini-graph, the template's execution frequency estimate is no longer correct. The mini-graph's coverage score is recomputed after the removed mini-graph's execution frequency is subtracted from the template's total execution frequency.

A working example of the greedy selection algorithm with template sharing is shown at the bottom of Figure 3.5. The start state shows the initial template pool with each template's coverage score. In the first iteration, `t2` and its members (`mg2` and `mg4`) are selected. The mini-graphs that overlap with `mg2`—`mg1` and `mg3`—are removed from the template pool. Because these two mini-graphs were the sole members of `t1` and `t3`, both of these templates are now empty and have no remaining coverage. At this point, there are no remaining templates in the template pool, and the algorithm halts. The total coverage is 350.

Good news and bad news. The good news is that the greedy algorithm with template sharing does, in fact, produce a higher coverage rate for a small MGT budget. If there were only room for a single mini-graph in the MGT, the coverage score of 350 is in fact superior to the basic greedy algorithm, which yields a coverage score of just 300 at an MGT budget of 1 entry. The bad news is that for a *larger* MGT budget, the greedy algorithm with template sharing produces a *lower* coverage rate.

With an MGT budget of 2 entries, for example, the coverage score for the greedy algorithm with template sharing remains 350; for the basic greedy algorithm, the coverage score increases to 450.

3.1.4 Backtracking to Include Larger Templates

The problem with the greedy algorithm with template sharing from the previous section is that it favors commonly occurring templates at the expense of larger, possibly less common—but still beneficial—templates. In this example, `mg3` (`ABCD`) is eliminated due to the selection of `mg3` (`ABC`). If the MGT could only hold a single template, the choice of just `t2` is correct. But if the MGT were large enough to support both templates, coverage would improve by supporting `t3` (*i.e.*, `mg3` (`ABCD`) as well. By simply rejecting all overlapping mini-graphs, the algorithm in Figure 3.5 no longer considers `mg3` once it has chosen `mg2`, despite the fact that it could still be useful.

The pseudo-code at the top of Figure 3.6 shows a slight modification of the algorithm to address this problem. Once again, algorithmic changes are shown in bold. In this version, smaller, more frequent mini-graph templates are chosen first, but their supersets remain in the selection pool in the event that the MGT is large enough to support both. (Smaller overlapping mini-graphs are always removed from the template pool because they will always have a lower coverage score due to size.) If the MGT budget permits, the larger mini-graph template can still be chosen. In such cases, the algorithm *backtracks*, un-doing its original decision to select the smaller mini-graph belonging to the more frequent template.

An example of the modified selection algorithm is shown at the bottom of Figure 3.6. During the first iteration, template `t2` has the highest coverage score (350) and is chosen first. When `mg2` (`ABC`) is selected, the smaller, overlapping mini-graph `mg1` (`AB`) is removed, thereby removing template `t1` from the pool. Overlapping mini-graph `mg3` (`ABCD`) is *not* removed from the pool because it is a strict superset of

Form template groups.
while (template pool not empty & budget not exceeded)

1. Choose T, template with largest coverage
2. Choose M, all mini-graph instances of T
3. **Un-choose S, all previously chosen subsets of M**
4. Remove all **non-superset** mini-graphs overlapping with instances of M from candidate pool
5. Update coverage of remaining templates in template pool

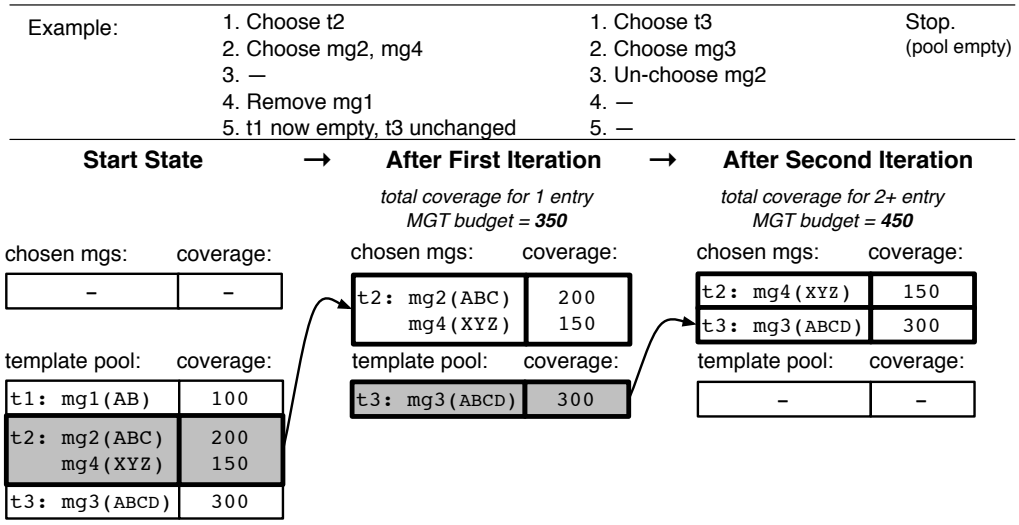


Figure 3.6: Greedy selection with template sharing and backtracking. Pseudo-code (top) and example (bottom). Modifications to previous algorithm shown in bold.

the chosen mini-graph. On the second iteration, template `t3` now has the highest coverage score (300) and it is selected. Even though `mg2` (`ABC`) was already selected, the algorithm backtracks; `mg2` (`ABC`) is removed from the chosen list, in favor of its superset mini-graph `mg3` (`ABCD`). At this point, there are no remaining templates in the template pool, and the algorithm halts. The total coverage under an MGT budget of 1 entry is 350 and under and MGT budget of 2 entries is 450. The greedy algorithm with template sharing and backtracking shows the best coverage rates for both the small and the larger MGT budget.

If enough larger mini-graphs etch away all of its mini-graph instances, `t2` might no longer be worth retaining. If the coverage score of `t2` ever dips lower than any of the templates in the template pool, it is returned to the template pool for re-selection. This step is easily added as a final step in the pseudo-code (not shown).

Coverage comparison. Figure 3.7 illustrates the behavior of the basic greedy algorithm and its two variants. The graph at the top of Figure 3.7 plots coverage rates for the three algorithms as the MGT budget drops from infinite, to 2048 entries, down to 4 entries. The results corroborate the both the intuition behind the algorithms and the toy example used thus far. The basic greedy algorithm exhibits superior coverage at large MGT budgets, but rapidly loses coverage as the MGT budget decreases. The greedy algorithm with template sharing shows improved coverage at small MGT budgets, but loses coverage relative to the greedy algorithm at large MGT budgets due to its unforgiving preference of template frequency over size. The greedy algorithm with template sharing and backtracking exhibits the best of both worlds, showing high coverage across the entire range of MGT budgets.

The graph at the bottom of Figure 3.7 shows the exact same data, but all coverage rates are relative to the basic greedy algorithm. The greedy with template sharing outperforms the basic greedy only when the MGT budget is small. The greedy algorithm with template sharing and backtracking has nearly the coverage of the basic greedy algorithm at large MGT budgets and superior coverage to greedy

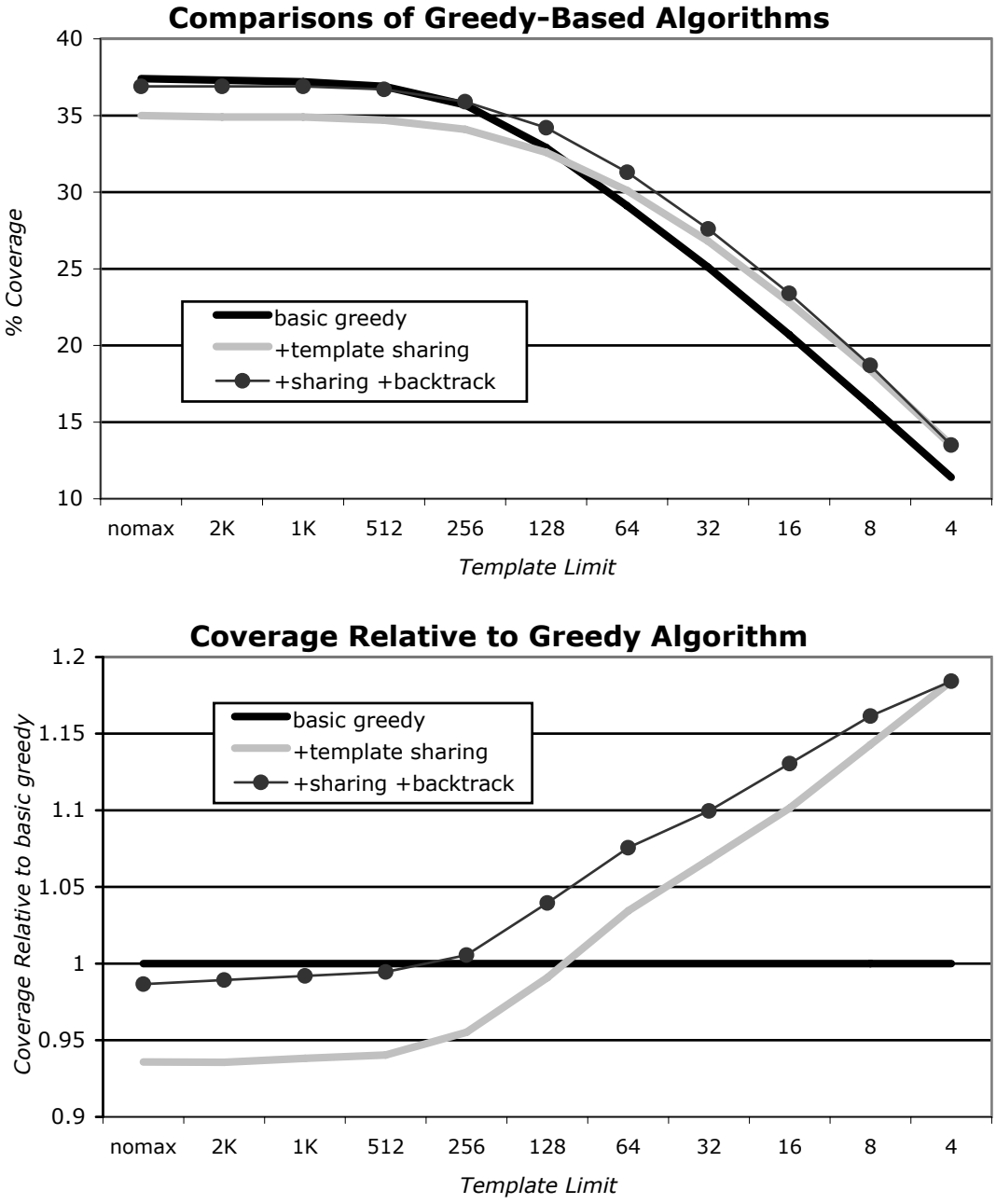


Figure 3.7: Variations of Greedy Algorithm.

algorithm with template sharing at all budget excepts when they converge at 4 entries. The basic greedy algorithm is slightly superior to the backtracking algorithm at high MGT budgets because the backtracking algorithm only keeps strict supersets in the template pool. The basic greedy algorithm can select larger, overlapping, non-supersets belonging to less frequent templates, whereas the backtracking algorithm cannot.

3.1.5 Exhaustive Selection

The basic greedy selection algorithm described in the previous section is a heuristic attempting to maximize coverage, but does not provably do so, even outside the context of template sharing. This section assesses the limitations of the basic greedy algorithm—without template sharing or backtracking—and addresses the question of whether a greedy approach could be improved upon with more comprehensive approaches. The two approaches explored are exhaustive search and integer programming formulation. Both techniques are provably optimal, *when they terminate*.

Exhaustive search. The obvious alternative to a greedy selection algorithm is an exhaustive search of all possible mini-graph combinations. A simple exhaustive search considers all $\sum_{k=1}^{t} \binom{n}{k}$ possible combinations of n candidate mini-graphs, where t represents the MGT budget. In other words, if there are n mini-graphs to select from, it explores every possible combination of t or fewer.

Each selected mini-graph, i, removes $(size(i)-1) \times frequency(i)$ dynamic instructions. The goal is to maximize the total number of dynamic instructions removed from a program:

$$Z = \forall i \sum (size(i) - 1) \times frequency(i)).$$

The combination with the largest value for Z is the optimal combination. The algorithm does not group mini-graphs into templates; none of the algorithms compared in this section do. For simplicity, the greedy algorithm is assessed in its most

basic form.

In the worst case, where $t = n$, the search space is exponential, *i.e.* $O(2^n)$. In practice, a technique that greatly reduces the search space of an exhaustive search is to prune conflicting mini-graphs mid-search. Once a mini-graph, i, is selected, the rest of the list of potential mini-graphs is pruned. Those that overlap with mini-graph i or those that no longer have a legal anchor position because mini-graph i and its predecessors were selected are pruned. At each selection point, then, the remaining list becomes smaller by more than just one, making the search space *potentially* smaller than 2^n in practice.

Algorithm	Problem Size		
	$\binom{20}{4}$	$\binom{40}{8}$	$\binom{60}{12}$
Greedy	10.2%	14.8%	18.2% *
Exhaustive	10.2%	14.9%	18.3% *

Table 3.1: Coverage Summaries: Greedy vs. Exhaustive. *-Results are shown for only 63 of 78 benchmarks because 15 benchmarks did not terminate on the Exhaustive algorithm after 2 days.

The benchmarks in this dissertation usually produce a candidate pool with thousands of mini-graphs. Performing an exhaustive search of such a large number of mini-graph combinations is simply not tractable. For this reason, the experiments of this section begin with a significantly smaller candidate pool. Table 3.1 shows the average coverage rates of the greedy and exhaustive selection algorithms achieved across 78 benchmarks. The candidate pools are fixed at sizes 20, 40, and 60, and the MGT budget is fixed at 4, 8, and 12, respectively. In each case, the candidate pool is comprised of the most frequently occurring n mini-graphs for each benchmark.

Average coverage scores for the greedy algorithm are nearly identical to those of the exhaustive search. Detailed results for all 78 benchmarks are shown in Figure 3.8. In a few cases, the greedy algorithm yields less coverage, but for all benchmarks across all three experiments the greedy coverage is at least 87% of the coverage of the exhaustive algorithm. Once the candidate pool reaches 60, only 63 of the 78

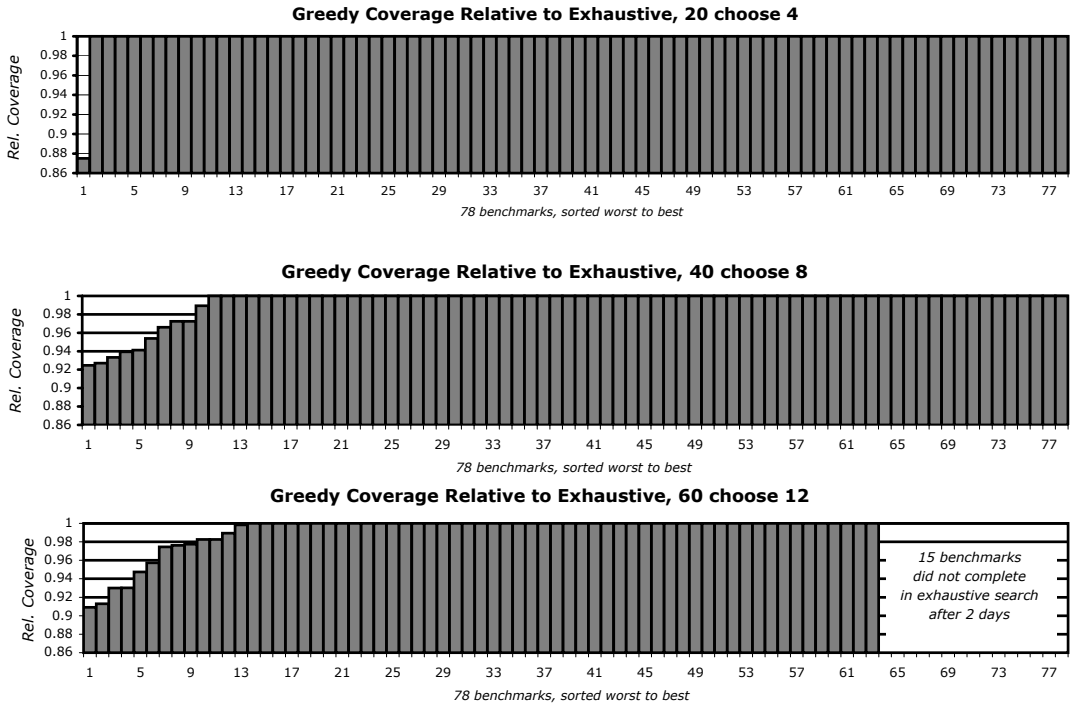


Figure 3.8: Individual Benchmarks: Greedy vs. Exhaustive Search. Greedy search relative to exhaustive search for 78 benchmarks.

benchmarks actually terminate within 2 days under the exhaustive search. For this reason, it is difficult to assess the effectiveness of the greedy algorithm in the context of more realistic or challenging search problems.

Mathematical programming-based search. An alternative but still optimal approach is to convert the greedy selection problem of maximizing coverage into an integer program. Unlike a greedy algorithm, an integer programming solver can offer incremental solutions that can be used in the case of long searches that might not otherwise terminate.

The selection problem can be written in the form of an integer programming problem as follows. There are n mini-graphs to choose from. Each mini-graph, i, has an indicator variable, x_i, that has the value 1 or 0 depending on whether the mini-graph i is selected or not. Each mini-graph, i, also has a utility variable, c_i, equal to the number of dynamic instructions that are removed by selecting mini-graph i. The goal is to maximize the total number of dynamic instructions removed from a program,

$$Z = c_1 x_1 + \ldots + c_n x_n. \tag{1.1}$$

The obvious solution to the following equation is to select all mini-graphs. Because overlapping mini-graphs conflict with one another, the equation is constrained as follows. For all mini-graphs $j, k, \ldots l$ that conflict with one another, only one can be selected. This is represented by the following constraint:

$$x_j + x_k + \ldots + x_l = 1. \tag{1.2}$$

The conflicts between overlapping mini-graphs are constructed just prior to performing the search. There are at most n conflict equations (at most one for each candidate mini-graph).

The restriction of the MGT budget is represented by the following constraint:

$$x_1 + \ldots + x_n \leq t. \tag{1.3}$$

where t represents the MGT budget. This prevents more than t mini-graphs from having an indicator variable of 1. In other words, no more than t mini-graphs will be selected.

The problem is solved by assigning a value of 0 or 1 to each variable x_i, while satisfying the list of constraints in (1.2) and (1.3) and maximizing the objective function (1.1).

The final constraint that is placed on mini-graph selection is that each mini-graph have a legal anchor point. Not having a legal anchor option is a property that is specific to the context of the other chosen mini-graphs. The only way to incorporate this constraint into the solver (without exhaustively considering every possible combination of mini-graphs) is to simply test the solution of the solver. In the event that a combination of mini-graphs $m, n, \ldots o$ does not have a legal anchor option for each mini-graph, the solver is run again after adding the following new constraint:

$$x_m + x_n + \ldots + x_o = 1 \tag{1.4}$$

Integer programming is known to be NP-hard, but can be approximated efficiently by relaxing it into a linear programming problem. The difference between an integer program and a linear program is that the former assigns integer values $\{0, 1\}$ to the variables x_i, while the latter allows real values within the interval $[0, 1]$. The solution of the linear program is converted into an integer solution by carefully rounding the values to either 0 or 1.

This dissertation uses the GNU Linear Programming Kit (GLPK) to find a feasible solution to the integer problem [64]. As with exhaustive search, some problems do find provably optimal solutions after days of computation. However, the GLPK performs a series of iterations; each iteration brings the solver closer to the optimal

solution. In cases of long-running problems, results can be harvested by terminating computation after a particular number of iterations have been reached and accepting the best found solution at that point. The higher this maximum is set, the closer the solution is to optimal, but the more likely it is that some programs will not terminate in a reasonable amount of time.

Algorithm	Problem Size		
	$\binom{100}{20}$	$\binom{200}{40}$	$\binom{400}{80}$
Greedy	20.1%	25.3%	29.2%
Integer Programming	21.4%	25.8%	29.4%

Table 3.2: Coverage Summaries: Greedy vs. Integer Programming.

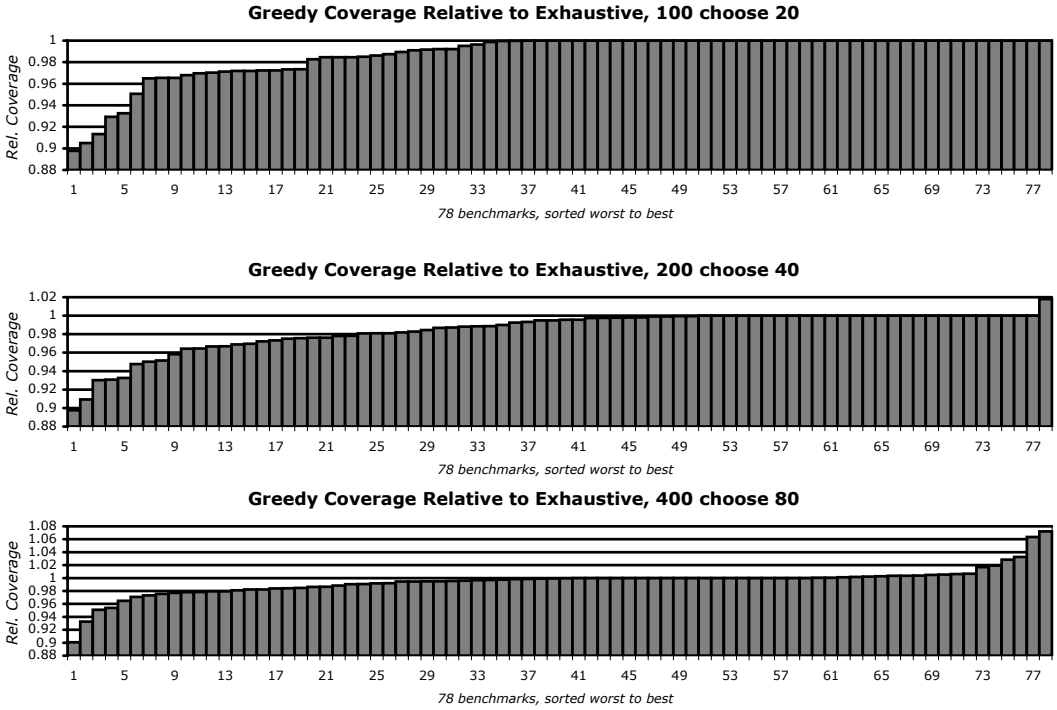


Figure 3.9: Individual Benchmarks: Greedy vs. Integer Programming. Greedy search relative to integer programming search for 78 benchmarks.

Table 3.2 shows the average coverage rates of the greedy and integer programming selection algorithms achieved across 78 benchmarks. The candidate pools are fixed at

sizes 100, 200, and 400, and the MGT budget is fixed at 20, 40, and 80, respectively. Once again, the candidate pool is comprised of the most frequently occurring n mini-graphs for each benchmark. The integer programming runs in this set of experiments were performed with a maximum number of iterations set to one million iterations. Although runs with no iteration limit completed for the $\binom{100}{20}$ experiment, runs for the $\binom{200}{40}$ and $\binom{400}{80}$ experiments did not complete within 24 hours.

Once again, the average coverage scores for the greedy algorithm are nearly identical to those of the integer programming search. Detailed results for all 78 benchmarks are shown in Figure 3.9. In a few cases, the greedy algorithm yields less coverage, but for all benchmark across all three experiments the greedy coverage is at least 88% of the coverage of the exhaustive algorithm. In some cases, the greedy algorithm actually out-performs the integer programming approach. This is the result of terminating the program after one million iterations—in some cases before a provably optimal solution was discovered. Interestingly, running this experiment several times could change the results. This is due to the fact that the integer programming solver incorporates a certain amount of randomness when it searches for solutions.

The experiments presented in this section show that a greedy selection algorithm is both effective and pragmatic. It is effective because it yields coverage rates competitive with provably optimal techniques. It is pragmatic because unlike both exhaustive and integer programming techniques, it reliably terminates in a matter of minutes. Furthermore, the greedy algorithm terminates within minutes for the actual selection tasks of this dissertation, not just the toy examples shown in this section. As a consequence, the remainder of the dissertation uses the greedy selection algorithm.

3.2 Introduction to Serialization

This section begins with a short evaluation and analysis of the coverage-maximizing selection algorithm presented in section 3.1. Initial results in Section 3.2.1 show that maximizing coverage gives rise to IPC penalties in many cases due to the introduction of serialization within a program. Section 3.2.2 explains the concept of serialization in detail. This yields way for the structural and subsequent slack-based selection algorithms of Sections 3.3 and 3.4, respectively.

3.2.1 Basic Coverage Maximizing Selection

This section presents a short evaluation and analysis of the coverage-maximizing selection algorithm. The evaluation tests the assertion that IPC improvement can be achieved by amplifying processor resources—without actually making them physically larger. To do so, it compares the IPC of an amplified, 3-wide mini-graph configuration to that of a 4-wide, non-mini-graph configuration, both relative to a 3-wide, non-mini-graph processor. The experimental evaluation throughout this dissertation follows the methodology introduced in Section 1.4 and the corresponding table in Figure 1.9.

Because ILP can be limited by many factors—including the nature of the benchmark itself—coverage rates do not translate to performance because amplification only improves performance in regions of high ILP. However, coverage rates are an important diagnostic tool because they indicate the amount of amplification present. Low coverage rates indicate low amplification and can be used to explain why a 3-wide mini-graph processor might not perform as well as a 4-wide processor. High coverage rates can explain why a 3-wide mini-graph processor might even out-perform a 4-wide processor.

Initial performance and coverage. The left graph in Figure 3.10 shows

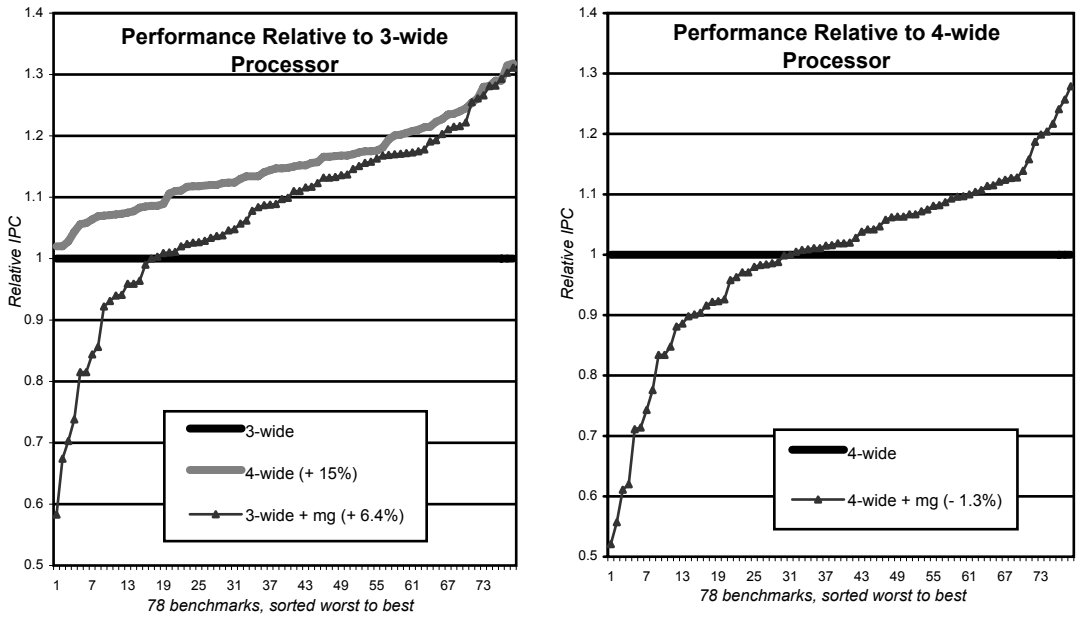


Figure 3.10: Amplifying vs. Increasing Resources. Left: Comparison of 3-wide mini-graph processor and 4-wide processor. Relative to a 3-wide processor. Right: Comparison of 4-wide mini-graph processor and 4-wide processor. Relative to a 4-wide processor.

performance—IPC relative to the 3-wide processor, whose own performance corresponds to the $y = 1$ axis—for mini-graphs selected by the coverage-maximizing selection algorithm. The 4-wide processor alone (no mini-graphs, light grey line) is on average 15% faster than the 3-wide processor. On the 3-wide configuration, mini-graphs achieve less than half of the 15% IPC gain achieved by creating a 4-wide machine, yielding an average improvement of 6.3% relative to the original 3-wide processor. However, individual results vary greatly. For some programs (on the right side of the graph), mini-graphs allow the 3-wide processor to out-perform the 4-wide processor. On 14 programs, it yields lower IPC than the original processor with no mini-graphs at all.

To provide additional insight, the right graph shows the IPC of the 4-wide processor, also enhanced with mini-graphs. Performance in this graph is relative to the 4-wide processor. A 4-wide processor enhanced with mini-graphs actually yields IPC *loss* for 28 programs (and an average IPC loss of 1.3% across all programs), relative to a standard 4-wide processor. For 14 of the benchmarks, the performance degradation seen at 4-wide is hidden on the 3-wide processor, which translates amplification to IPC at a high rate. On the 4-wide processor, on which amplification provides fewer IPC benefits and penalties are more exposed, these slowdowns are apparent.

This initial experiment indicates that from an IPC standpoint, mini-graph processing driven by coverage maximizing selection is not a robust alternative to physical increases in superscalar width and window size. The mini-graph IPC loss seen in both graphs can be attributed to mini-graph serialization. After explaining the concept and causes of serialization, the following sections show how to select mini-graphs not only to maximize coverage, but also to minimize the IPC degradation associated with serialization.

3.2.2 What is Serialization?

A dynamically scheduled processor ostensibly executes singleton instructions in data dependence order. Mini-graph-induced serialization is an artificial dependence between two singleton instructions that is created when both instructions are placed in the same mini-graph. There are two forms of serialization. In both cases, an artificial dependence is created between two instructions. Serialization can degrade performance, even to the point of overwhelming the benefits of mini-graph processing.

External serialization, the more frequent and destructive form, introduces a new dependence between the first constituent in a mini-graph and some instruction outside the mini-graph; this outside instruction produces an input to the mini-graph, but this is not an input to the first constituent. External serialization occurs because a mini-graph cannot issue until all of its external register inputs are available. Most forms of instruction fusion—not just mini-graphs—are subject to external serialization.

Micro-op fusion, however, is *not* one of the forms of instruction fusion that suffers from external serialization. Avoiding serialization is, in fact, cited as one of the reasons the store-address/store-data macro instruction was split in the first place [38]. By fusing operation pairs into single, expanded entries, micro-op fusion amplifies the capacity of the reservation station. Because the operations themselves still issue separately, however, scheduling bandwidth is *not* amplified. Mini-graphs, like many other forms of fusion, expose themselves to external serialization by issuing aggregates atomically. They do so because of the desire to extend resource amplification to scheduling bandwidth.

Internal serialization, a less dominant form, is a dependence between an instruction in a mini-graph and a previous independent instruction in the same mini-graph. Internal serialization occurs in mini-graphs because mini-graph constituents execute

in series. Most—but not all—forms of aggregation are subject to internal serialization.

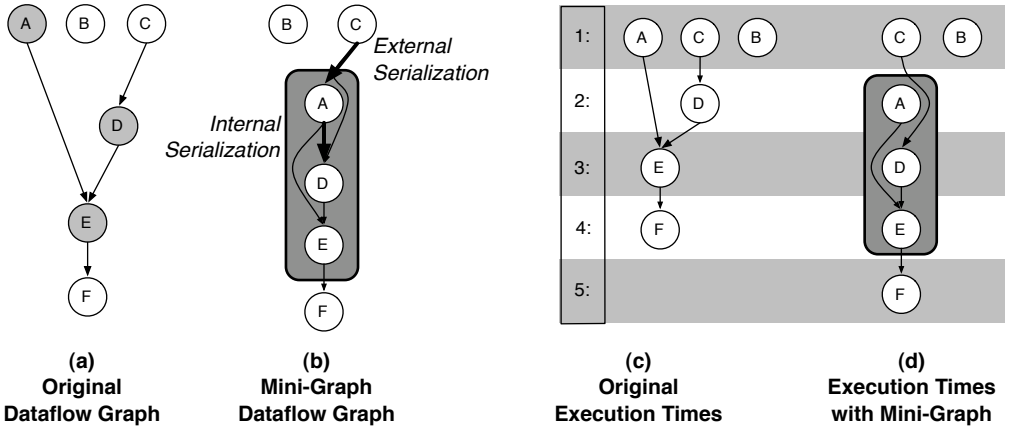


Figure 3.11: Serialization Effects in Mini-Graphs

Figure 3.11 shows an example of serialization, beginning with the original dataflow graph of six abstract instructions `A-F` in Figure 3.11a. The dataflow graph after forming mini-graph `ADE` is shown in Figure 3.11b. External serialization requires the head of the mini-graph to wait for all inputs, creating a new dependence edge between instructions `A` and `C`. Internal serialization requires all mini-graph constituents to wait for one another, creating a new dependence edge between instructions `D` and `A`.

Figure 3.11c shows a singleton execution of these instructions. Figure 3.11d shows the execution of the same instructions, but with instructions `ADE` aggregated into a mini-graph. Notice, each serializing edge induces a delay on the corresponding instruction: `A` and `D` are both delayed by 1 cycle. However, the total delay on the output of the mini-graph, `E`, is 1 cycle. Here, the external serialization *masks* the internal serialization. The 6-instruction sequence `A-F` previously completed in 4 cycles; with mini-graphs, it uses far fewer resources, but completes in 5 cycles.

Experience with mini-graphs shows that internal serialization is often masked by external serialization. This suggests that, at least for mini-graphs, a constituent

execution model that simplifies implementation at the cost of adding internal serialization is a reasonable design choice.

Mini-graphs are not alone in their vulnerability to both forms of serialization. Most aggregation schemes are vulnerable to external serialization [23, 83, 89, 112], and many are also vulnerable to internal serialization as well [58, 89]. However, mini-graphs suffer from serialization more than most other aggregation techniques when they do not provide a direct latency-reduction benefit that can directly counter-act serialization-induced delay (*e.g.*, interlock-collapsing ALU Pipelines). By focusing on resource amplification, mini-graphs require wholesale (rather than opportunistic) aggregation, increasing the probability of harmful mini-graphs.

3.3 Structural Selection Algorithms

The basic mini-graph selection algorithm described in Section 3.1.1 starts by identifying an initial pool of static mini-graph candidates. From there, it selects mini-graph templates in order to maximize dynamic coverage (*i.e.*, resource amplification). An abstract view of the original selection process is shown in Figure 3.12a. Shading indicates serialization. Numbers are hypothetical and used for explanatory purposes only. Step 1 begins with 42 mini-graphs, some of which overlap. Step 2 greedily selects non-overlapping mini-graphs, resulting 19 selected mini-graphs. The selected set contains mini-graphs with and without serialization.

Static, structural pruning. Figure 3.12b depicts the 3-step static, structural pruning algorithm. Unshaded mini-graphs are structurally benign and are always included in the initial selection pool of every selection algorithm. In Step 1.5, structural pruning removes from this pool all mini-graphs with any structural potential for serialization (according to an algorithm's definition of potential). As a result, the greedy algorithm has only 20 overlapping mini-graphs to select from, resulting in a total of 12 selected mini-graphs.

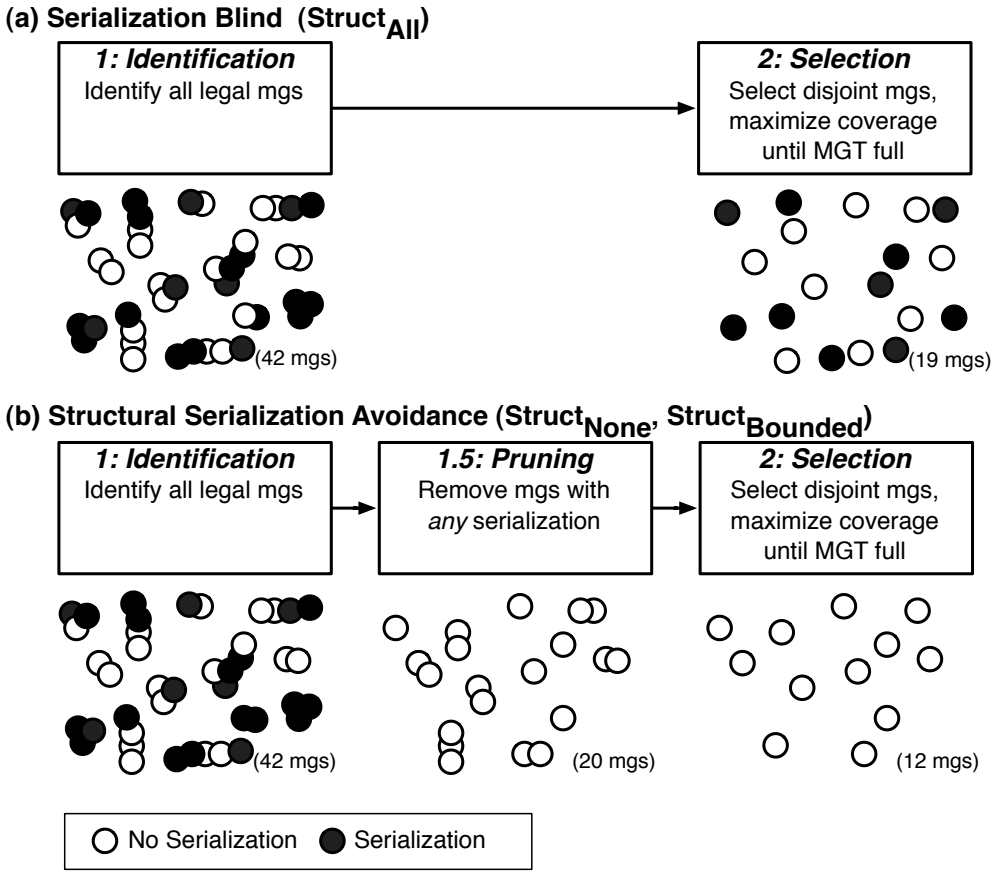


Figure 3.12: Basic Selection vs. Structural Pruning Selection. (a) Serialization Blind (b) Structural Serialization Avoidance.

This section presents two structural mini-graph selection algorithms that are performed statically in software before the program runs. The first, $\text{Struct}_{\text{None}}$, is a naive selection algorithm that simply forbids potential serialization entirely. The second, $\text{Struct}_{\text{Bounded}}$, is a less conservative algorithm that forbids only a particular type of serialization, called *bounded serialization*.

3.3.1 $\text{Struct}_{\text{None}}$

This section briefly compares two naive responses to the problem of serialization. The first response is to ignore the problem and continue to maximize coverage. This algorithm (which has already been introduced) is hereafter referred to as $\text{Struct}_{\text{All}}$, because it admits *all* possibly serializing mini-graphs.

The second response—introduced in this section—is to avoid serialization entirely by only selecting non-serializing mini-graphs. By virtue of their dataflow shape, some mini-graphs are simply not vulnerable to serialization. This can be easily inspected; the first instruction of a structurally benign mini-graph depends on each and every external register input of the mini-graph. This algorithm is called $\text{Struct}_{\text{None}}$, because it admits *no* possibly serializing mini-graphs.

Figure 3.13 compares the IPC (left) and coverage (right) of both algorithms. The performance graph shows IPC relative to the 3-wide processor and also shows results for the 4-wide, non-mini-graph processor for comparison.

$\text{Struct}_{\text{All}}$. $\text{Struct}_{\text{All}}$ is an aggressive selector that admits *all* potentially serializing mini-graphs into the starting pool. This selector—the original selector introduced in Section 3.1—is serialization blind. It maximizes coverage without any thought to potential slowdowns induced by serializing mini-graphs. Naturally, $\text{Struct}_{\text{All}}$ achieves the highest coverage rates (of about 37%).

Without serialization, the high amplification rates of $\text{Struct}_{\text{All}}$ should allow mini-graphs to simulate—in terms of performance—the increase in processor resources from a 3-wide fetch/issue/commit to a 4-wide fetch/issue/commit and from 20 issue

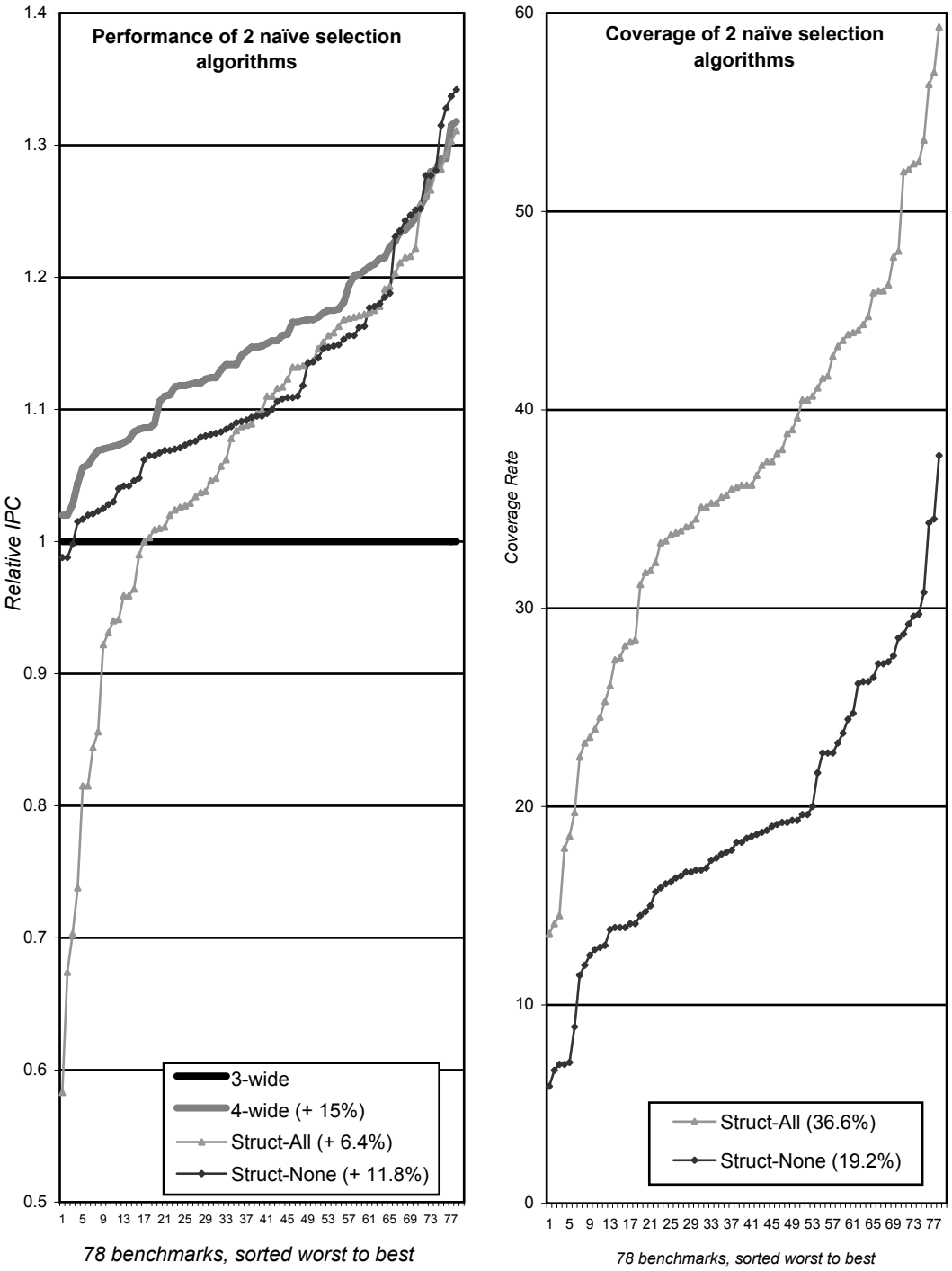


Figure 3.13: Ignoring vs. Forbidding Serialization. Left: Comparison of $\text{Struct}_{\text{All}}$ and $\text{Struct}_{\text{None}}$ on a 3-wide mini-graph processor and 4-wide processor. Relative to a 3-wide processor. Right: coverage of $\text{Struct}_{\text{All}}$ and $\text{Struct}_{\text{None}}$ algorithms.

queue entries and 56 rename registers to 30 and 80, respectively; an increase that typically results in a 15% IPC improvement, on average. Unfortunately, $\text{Struct}_{\text{All}}$ admits a small number of pathological serializing mini-graphs that degrade the IPC of 20% of all programs.

$\text{Struct}_{\text{None}}$. $\text{Struct}_{\text{None}}$ is a conservative selector that admits *no* potentially serializing mini-graphs into the starting pool. This selector sits on the opposite end of the selector spectrum from $\text{Struct}_{\text{All}}$, assuming all mini-graphs that appear structurally vulnerable to serialization are harmful to performance. It removes all possibilities of performance loss, but with that removes a good deal of coverage as well. By cutting coverage in half of that offered by $\text{Struct}_{\text{All}}$, $\text{Struct}_{\text{None}}$ makes it impossible to compete IPC-wise with a 4-wide processor.

$\text{Struct}_{\text{None}}$ mini-graphs (diamond) produce better average IPC, achieving an 11.8% improvement over the 3-wide machine. IPC gains are also more consistent. $\text{Struct}_{\text{None}}$ almost always outperforms the non-mini-graph processor. However, for about half the programs, it provides less IPC than $\text{Struct}_{\text{All}}$. The key here is coverage. $\text{Struct}_{\text{All}}$ yields coverage rates from 15% to 60% (37% on average). By conservatively rejecting all mini-graphs with serialization potential, $\text{Struct}_{\text{None}}$ has only half this coverage, ranging from 6% to 38% (19% on average).

The case for a serialization-aware "hybrid" scheme. The different shapes of the $\text{Struct}_{\text{All}}$ and $\text{Struct}_{\text{None}}$ S-Curves and their "cross-over" behavior 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 IPC of $\text{Struct}_{\text{All}}$ when amplification is at a premium and $\text{Struct}_{\text{None}}$ when amplification is ineffective and serialization dominates. However, a hybrid scheme that is intelligently serialization-aware should be able to consistently outperform both $\text{Struct}_{\text{None}}$ and $\text{Struct}_{\text{All}}$, as it should be able to make coverage vs. serialization decisions on a per mini-graph basis. The expected IPC of this hybrid is approximately the IPC of the 4-wide, non-mini-graph processor

(square of Figure 3.13), which has the IPC these experiments have been targeting all along.

3.3.2 $\text{Struct}_{\text{Bounded}}$

$\text{Struct}_{\text{All}}$ and $\text{Struct}_{\text{None}}$ are two extreme structural approaches to dealing with potentially serializing mini-graphs. $\text{Struct}_{\text{Bounded}}$ represents a heuristic compromise. The observation behind $\text{Struct}_{\text{Bounded}}$ 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, $\text{Struct}_{\text{Bounded}}$ 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.

Bounded vs. unbounded serialization. Figure 3.14 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. Figure 3.14a shows an example. Both instructions `A` and `B` are ready at the same time, but `B` waits for `A`, delaying the register output of the mini-graph by the execution latency of `A`.

In the context of external serialization, delay is bounded if the serializing input is "upstream" from the mini-graph register output. The mini-graph in Figure 3.14b has bounded serialization. Even if the serializing input is ready n cycles after the input to the first instruction in the mini-graph, the delay on the mini-graph's register output (`B`) is equal only to the latency of instruction `A`. This is because in a singleton execution, `B` is dependent on and ultimately waits for the serializing input anyway. In contrast, the slightly different mini-graph in Figure 3.14c 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

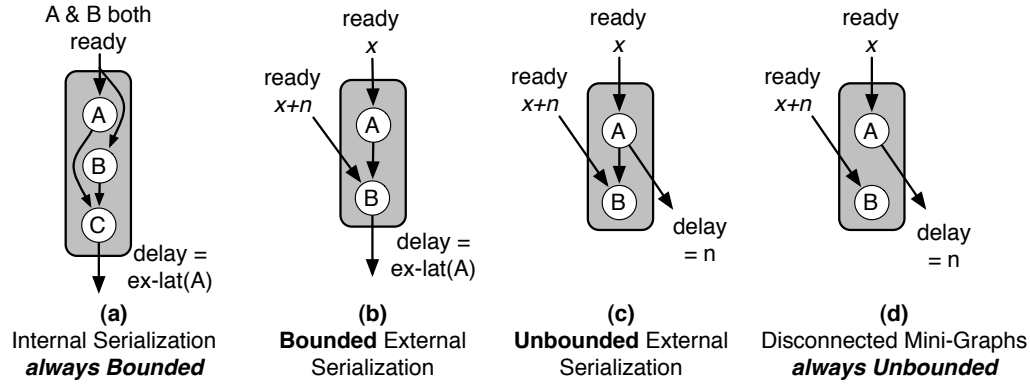


Figure 3.14: Bounded vs. Unbounded Serialization.

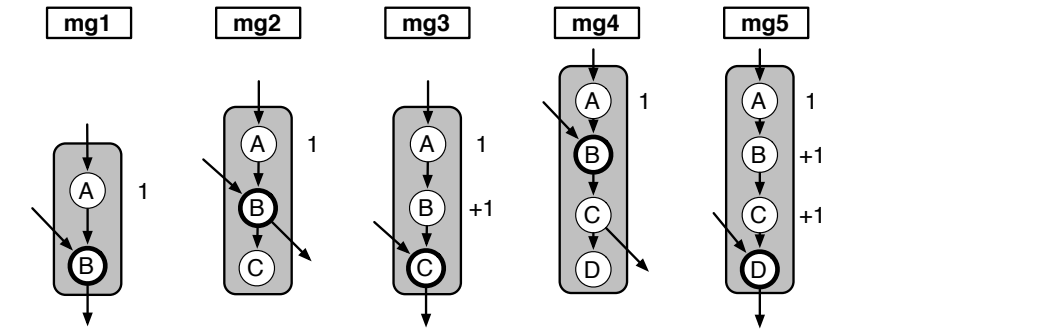
register output is delayed by n cycles. Here, in a singleton execution, `A` never waits for `B`'s input.

Disconnected mini-graphs (shown in Figure 3.14d) are a special case of the unbounded serialization in Figure 3.14c. Once again, if the instruction producing the mini-graph register output (`A`) is required to wait an additional n cycles for the input to a second independent instruction (`B`), the mini-graph's register output is delayed by n cycles.

Store and branch outputs. Although a mini-graph has only a single register output, it 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 if they forward their values to younger in-flight loads and branches act as outputs when they are mis-predicted. $\text{Struct}_{\text{Bounded}}$ bounds the delay on register and memory outputs. In practice, however, the register output plays the dominant role. This is because the determining factor in whether a mini-graph has bounded or unbounded serialization is the position of the most "upstream" output relative to the position of the serializing input (Figure 3.14b vs. Figure 3.14c). Branches cannot play a role here because they are always mini-graph terminal and therefore control outputs can never be upstream of anything. Stores are *almost* always located at the end of a

mini-graph and are *never* upstream of a register output. Only in the rare case of a mini-graph that has a store *and* a branch and *no* register output does a memory output affect the categorization of serialization as unbounded. The common case is shown in Figure 3.14c. Instruction `B` is either a store or a branch. Yet it is the register output of instruction `A` that categorizes this serialization as unbounded. Here, the control/memory output plays no role whatsoever. Even though $\text{Struct}_{\text{Bounded}}$ considers delays on both memory and register outputs, in practice only the register output matters.

By allowing all mini-graphs with bounded serialization and no mini-graphs with unbounded serialization, $\text{Struct}_{\text{Bounded}}$ accepts strictly more mini-graphs than $\text{Struct}_{\text{None}}$ and strictly fewer than $\text{Struct}_{\text{All}}$. 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 mis-predicted branch.



1	1	2	1	3	Output Delay
✓	✓	×	✓	×	$\text{Struct}_{\text{Bounded}}1$
✓	✓	✓	✓	×	$\text{Struct}_{\text{Bounded}}2$
✓	✓	✓	✓	✓	$\text{Struct}_{\text{Bounded}}3$
✓	✓	✓	✓	✓	$\text{Struct}_{\text{Bounded}}$

Figure 3.15: Serialization Bounded by Various Cycles. Mini-graphs with delays of d are accepted by $\text{Struct}_{\text{Bounded}}$x where $d <= x$. $\text{Struct}_{\text{Bounded}}$ accepts all mini-graphs with bounded delay.

Struct$_{\text{Bounded}}$N. In general, static analysis can bound delays to any number of cycles up to the execution latency of the mini-graph minus one. In the case of the mini-graphs of this dissertation, the maximum delay is 6 cycles. (Mini-graphs can have an execution latency of up to 7 cycles; if the serializing input feeds the last instruction in a 7-cycle mini-graph, this final instruction waits 6 cycles beyond when its input arrives before it can execute.) With this in mind, Struct$_{\text{Bounded}}$ is actually the most lenient instance of a more general selector, Struct$_{\text{Bounded}}$N. Struct$_{\text{Bounded}}$N is a selector that accepts only mini-graphs with a potential serialization delay of n cycles. For this dissertation, in which mini-graphs are supported up to 7 cycles, Struct$_{\text{Bounded}}$, is really an instance of Struct$_{\text{Bounded}}$6. (Incidentally, if one were to ignore internal serialization, Struct$_{\text{None}}$ is actually an instance of Struct$_{\text{Bounded}}$0.)

Stricter selectors that allow bounded serialization with a tighter delay bound can be created by choosing a smaller value for N. Figure 3.15 shows some examples. Instructions `A-D`are single-cycle operations. Mini-graphs 1 through 5 have various output delays. All exhibit bounded serialization and are therefore accepted by Struct$_{\text{Bounded}}$. A delay of d cycles is accepted by Struct$_{\text{Bounded}}$x, where $d <= x$. In practice, because most mini-graphs are only two or three cycles in length, it is expected that Struct$_{\text{Bounded}}$N will converge to the coverage rates of Struct$_{\text{Bounded}}$ at around Struct$_{\text{Bounded}}$4.

Performance and coverage. Figures 3.16- 3.17 show the performance and coverage results of Struct$_{\text{Bounded}}$N for values of N from 1 to 3 as well as for Struct$_{\text{Bounded}}$, which is Struct$_{\text{Bounded}}$6 for 7-cycle mini-graphs. Figure 3.16 shows IPC of 3-wide mini-graph processors run with various instantiations of Struct$_{\text{Bounded}}$N all relative to a 3-wide processor (at y=1). Also shown is the 4-wide non-mini-graph processor which shows a relative IPC improvement of 15%.

The IPC improvement offered by the Struct$_{\text{Bounded}}$N algorithms vary from 14.2% to 17.0%. In general, the more conservative the algorithm, the better the IPC. The converse is naturally true for coverage. Coverage ranges from 25.7% to 29.1%, with

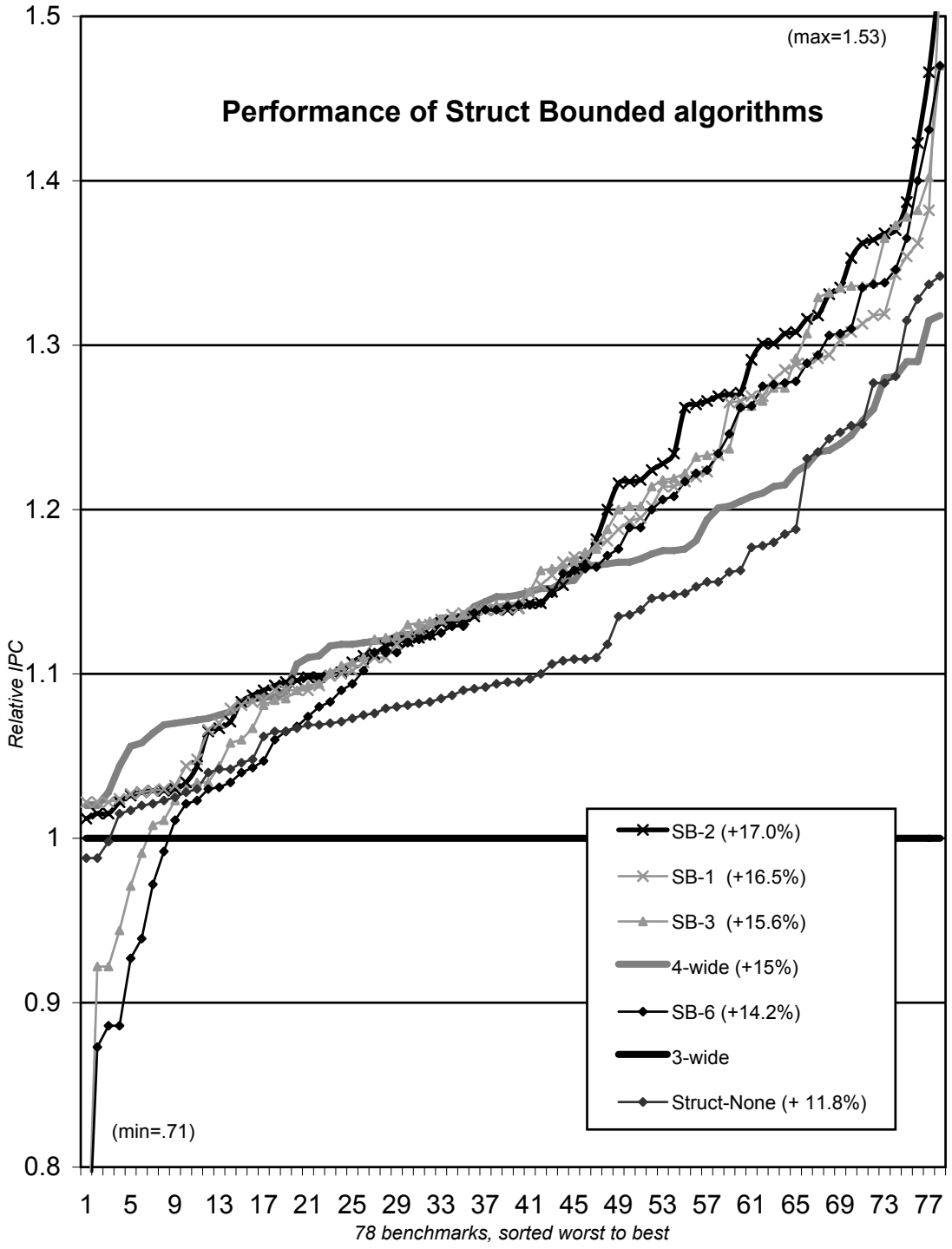


Figure 3.16: $\text{Struct}_{\text{Bounded}}$. IPC relative to a 3-wide processor of $\text{Struct}_{\text{Bounded}}$N, for N = 1,2,3,6

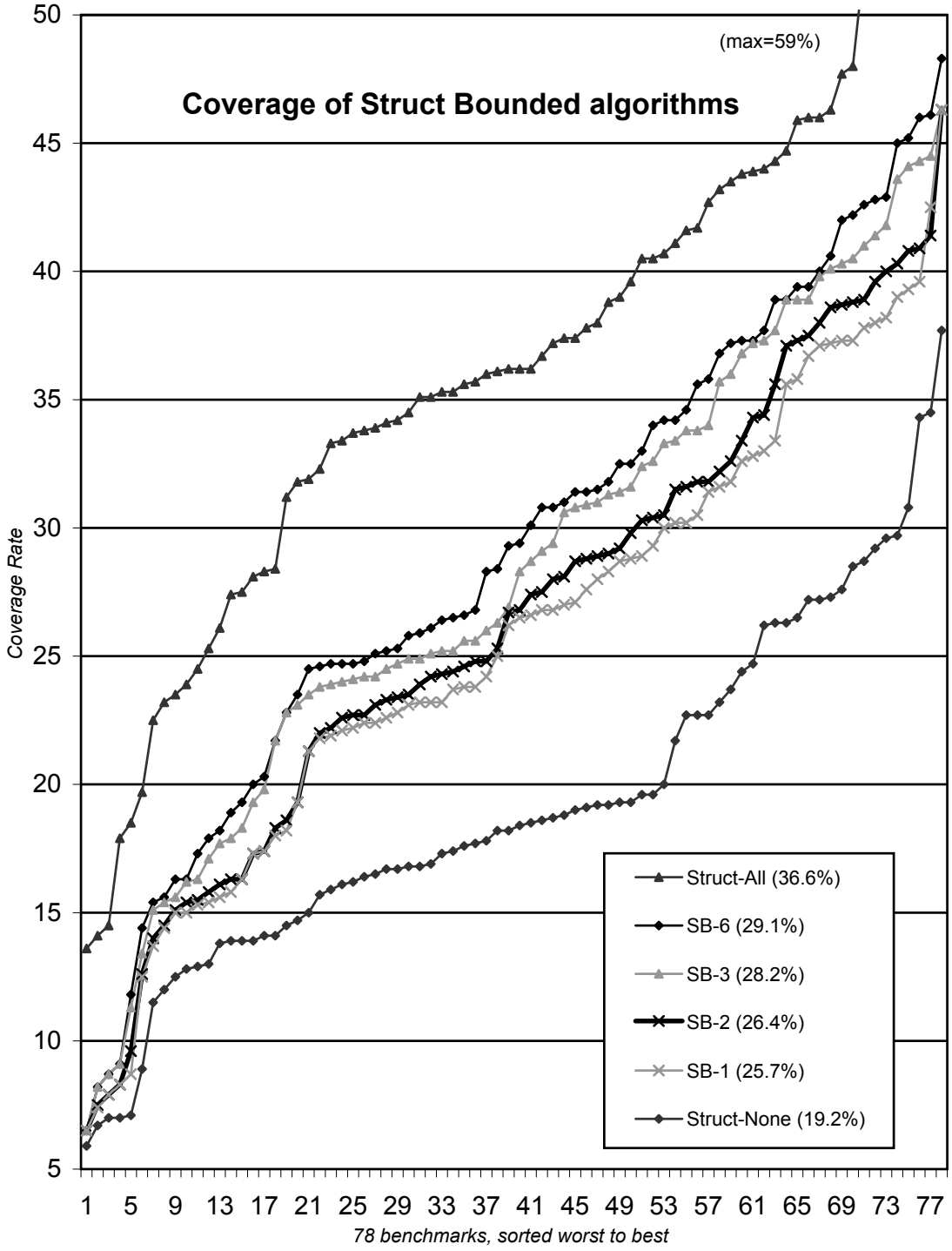


Figure 3.17: $\text{Struct}_{\text{Bounded}}$. Coverage of $\text{Struct}_{\text{Bounded}}$N, for N = 1,2,3,6

the more conservative algorithms pruning more mini-graphs, resulting in lower coverage. In the cases where IPC improves significantly (the right side of the graph), all $\text{Struct}_{\text{Bounded}}$N algorithms behave similarly. In the worst cases, the conservative algorithms distinguish themselves as clearly preferable. $\text{Struct}_{\text{Bounded}}3$ and $\text{Struct}_{\text{Bounded}}(\text{Struct}_{\text{Bounded}}6)$ both exhibit IPC losses of over 25%, with approximately 10% of benchmarks exhibiting some form of IPC degradation. (As expected, results for $\text{Struct}_{\text{Bounded}}4$ were nearly identical to that of $\text{Struct}_{\text{Bounded}}6$; these data are not shown in order to make the graph more legible.)

$\text{Struct}_{\text{Bounded}}1$ and $\text{Struct}_{\text{Bounded}}2$ are clearly the best of the $\text{Struct}_{\text{Bounded}}$N algorithms. Although they have the lowest coverage, they exhibit the highest IPC improvement, and the best robustness. $\text{Struct}_{\text{Bounded}}2$ behaves like a shifted version of $\text{Struct}_{\text{None}}$ (thick grey line); their coverage and performance curves have the same basic shapes and slopes. $\text{Struct}_{\text{Bounded}}2$ provides more coverage and amplification than $\text{Struct}_{\text{None}}$ (29% vs. 19%), but significantly better relative IPC (17% vs. 12%). By avoiding unbounded serialization, the $\text{Struct}_{\text{Bounded}}$N algorithms avoid the performance pathologies of mini-graph serialization. Whereas $\text{Struct}_{\text{All}}$ induces slowdowns for 16 programs on a 3-wide processor, the $\text{Struct}_{\text{Bounded}}$N algorithms induce none. By allowing 2 cycles of bounded delay instead of just 1, $\text{Struct}_{\text{Bounded}}2$ manages to offer slightly more coverage (26.4% vs. 25.7%) which translates into slightly better IPC (17.0% vs. 16.5%). The remainder of this dissertation discusses only $\text{Struct}_{\text{Bounded}}2$ as it is the best of the $\text{Struct}_{\text{Bounded}}$N algorithms.

3.4 Slack-Based Selection Algorithms

The challenge in detecting problematic mini-graphs is that serialization and serialization-induced performance loss cannot be deduced by inspecting dataflow structure. Not all mini-graphs with the potential for serialization are actually delayed by the serializing input. In some cases, the potentially serializing input

is always ready first, and no delay occurs. And even if delay is induced, it might be masked by other delays and have no performance effect. These mini-graphs manifest serialization but don't degrade IPC because the delayed output is "off the critical path." The algorithms in this section attempt to quantify these structural unknowns in order to better determine which mini-graphs to prune.

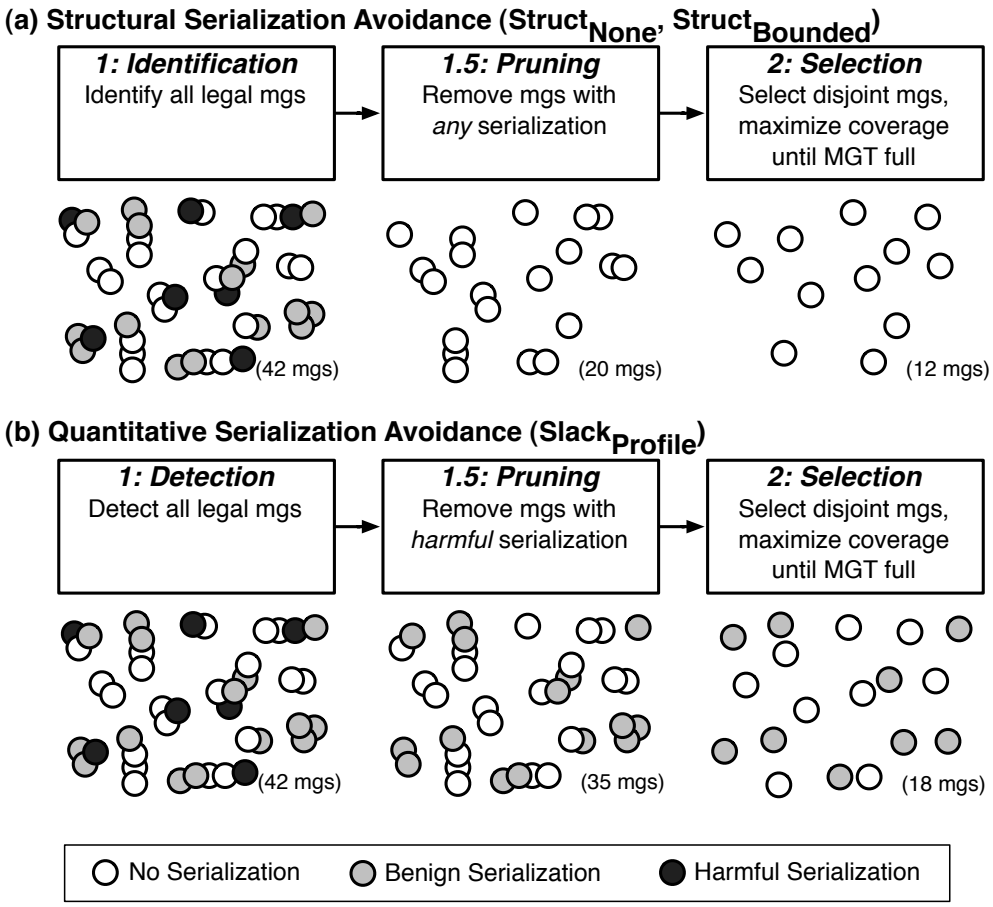


Figure 3.18: Structural vs. Slack-Based Pruning Selection. (a) Structural Serialization Avoidance. (b) Static, Slack-based Serialization Avoidance.

The main problem with structural heuristics is that they remove mini-graphs with essentially benign serialization. Figure 3.18a illustrates. If the majority of mini-graphs with serialization are actually benign, then removing all serializing mini-graphs from the selection pool is too conservative. The slack-based algorithms in

this chapter attempt to detect mini-graphs with harmful serialization only. Ideally, the algorithms function as shown in Figure 3.18b. Step 1.5 removes *only harmful* mini-graphs from the initial pool of 42 identified mini-graphs, leaving 35 mini-graphs to choose from. From here, greedy selection resumes (Step 2), yielding a total of 18 mini-graphs.

The key here is to find a pruning technique that can most accurately distinguish between the benign and harmful cases. Techniques that over-prune have lower coverage. Techniques that under-prune have IPC loss. The slack-based selection algorithms presented in this section are designed to distinguish between harmful and benign instances of serialization by paying attention to the criticality of potentially delayed instructions. The first algorithm is performed statically in software before the program runs. The second algorithm is dynamic and is performed in hardware at runtime.

3.4.1 $\text{Slack}_{\text{Profile}}$

$\text{Slack}_{\text{Profile}}$ is a static selection algorithm that uses profiles—information gathered during the actual run of a program—to determine both whether the formation of a mini-graph induces a delay and whether this delay can be absorbed by the program with no performance penalty. The ability to absorb delay is formalized by *local slack* [32]. An instruction's local slack is the number of cycles by which it can be delayed without delaying any consumer. For example, if an instruction can be delayed 2 cycles before any of its consumers are delayed, that instruction is said to have a local slack of 2 cycles. A specific example is shown later in this section.

$\text{Slack}_{\text{Profile}}$ begins with a profiled singleton execution schedule that 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. This section also discusses $\text{Slack}_{\text{Profile}}$'s profiling

support and the rationale for its use of local, rather than global, slack.

Quantifying mini-graph induced serialization delay. (Rules 1-3.) 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, however, an instruction waits for all inputs to the mini-graph, even those it was independent of in singleton form. This phenomenon is encapsulated in the following rule:

Rule 1, External Serialization :

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

The issue time of the first instruction of a mini-graph is determined by the ready time of all inputs to the mini-graph as well as the issue time of this instruction according to the singleton execution schedule. Including the issue time of the instruction as a singleton incorporates non-data dependences (*e.g.*, structural dependences) that may continue to play a role in an instruction's issue time when in mini-graph form.

The issue time of subsequent instructions in a mini-graph is determined by the issue time of the previous mini-graph instructions. This phenomenon is encapsulated in the following rule:

Rule 2, Internal Serialization :

$$Issue_{MG}(n) = Issue_{MG}(n-1) + ExLat(n-1)$$

The issue time of all non-initial mini-graph instructions is simply the time the previous instruction issued plus the execution latency of that previous instruction. The execution latency of each instruction is known *a priori*. In the framework of this dissertation, all mini-graph constituent execute in 1 cycle, excepting loads which execute in 3 cycles. More sophisticated frameworks could also incorporate knowledge about load misses to provide a more accurate estimate. If a mini-graph processor had a form of execution latency reduction, this should be reflected in the estimate for $ExLat$ for better accuracy. Other than this refinement, the entire algorithm remains unaffected in the presence of latency reduction hardware.

For each mini-graph candidate, $\text{Slack}_{\text{Profile}}$ uses Rules 1 and 2 to determine the issue time of each mini-graph constituent. Finally, for each instruction in the mini-graph candidate, $\text{Slack}_{\text{Profile}}$ uses the following rule to determine whether delay is induced by mini-graph formation:

Rule 3, Instruction Delay :

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

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.

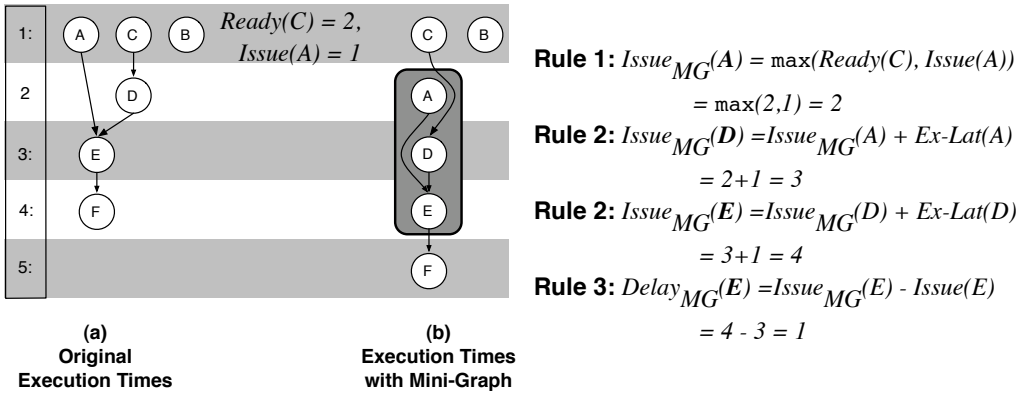


Figure 3.19: $\text{Slack}_{\text{Profile}}$. Calculating the delay on instruction E induced by the formation of mini-graph ADE.

Figure 3.19 steps through a delay calculation. $\text{Slack}_{\text{Profile}}$ starts with the singleton execution schedule in Figure 3.19a and calculates the mini-graph execution schedule in Figure 3.19b. The quantity of interest is the delay induced on instruction `E` by the formation of mini-graph `ADE`. The mini-graph has one external input, from instruction `C`, which is ready at cycle 2. $\text{Slack}_{\text{Profile}}$ uses Rule 1 to calculate the new issue time of instruction `A`, Rule 2 (twice) 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. (Rule 4.) Instruction delay degrades IPC only if it cannot be "absorbed" by consuming instructions. In Figure 3.19, instruction `A` has 1 cycle of local slack; it could be delayed 1 cycle without delaying `E`.

$\text{Slack}_{\text{Profile}}$ uses per-static instruction local slack estimates to calculate whether a given mini-graph, if formed, degrades IPC. It does so using the following rule:

Rule 4, Performance Degradation :

$$Degrade_{MG}(0) = \bigvee_{i \in mg-outputs} (Delay_{MG}(i) > Slack(i))$$

A mini-graph degrades performance if for any of its outputs, the induced delay is greater than that output's local slack. The profiler provides slack information for stores and branches as well as instructions producing register values, so all three forms of outputs can be explicitly considered. It is important to include slack on stores and branches in the algorithm. To illustrate, a single mini-graph that induced a 1-cycle delay on a hard-to-predict branch (*i.e.*, a branch with no slack) in a frequently executed loop in *adpcm.encode* single-handedly degraded performance by 3%!

In the example in Figure 3.19, the formation of mini-graph `ADE` delays `E` by 1 cycle. `ADE` is rejected because `E` has a local slack of 0 cycles, and its delay is propagated to `F`. If, instead, `E` had a few cycles of local slack—perhaps `F` were an easy to predict branch—then `E`'s delay is easily absorbed by the program and the mini-graph is not rejected.

Profiling support. As explained, $\text{Slack}_{\text{Profile}}$ requires local slack estimates, $Issue$ times, and *Ready* times. The slack profiling tool—in the case of this dissertation, a simulator—first generates the *edge local slack*, the slack between any two instructions, according to the following definition:

$$\forall ij, j \in consumers(i),\ EdgeLocalSlack(ij) = Issue(j) - Ready(i)$$

Suppose instruction `A` feeds instruction `B` and instruction `A` completes (*i.e.*, its result is ready) at time t. If instruction `B` issues at time $t + 2$, then the slack on the edge `AB` is 2 cycles. Instruction `A` could be delayed 2 cycles without delaying `B`.

Once all of the edge local slack values have been calculated for a particular instruction, the local slack of each static instruction is calculated according to the following definition:

$$\text{Local Slack(i)} = \min_{j \in consumers(i)} EdgeLocalSlack(ij)$$

Suppose an instruction `A` feeds three instructions `B`, `C`, and `D`, with edge local slack values of 1, 2, and 4. The local slack on instruction `A` is 1 cycle, because instruction `A` could be delayed only 1 cycle before delaying a consumer.

It is important to notice that the issue and ready times required by $\text{Slack}_{\text{Profile}}$ are, by definition, already generated and used in the course of calculating local slack. Acquiring these times doesn't require heavier profiling, just more verbose profiler output. 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).

Calculating issue and ready times from local slack estimates. Although modifying a local slack profiler to output issue and ready times requires no extra computation, there is also a clever way to simply calculate the $Issue$ and $Ready$ times from local slack profiles without requiring any additional output. This method only works for mini-graphs that are connected in a dataflow graph, and is consequently not a general-purpose solution. It is, however, worth introducing as an alternative to modifying a slack profiler.

A good visual analogy is as follows. Think of a connected dataflow graph as being implemented with physical links like ropes, where the length of each link is the amount of slack on the corresponding edge. Now think of terminal outputs in

this graph (instructions that have no outputs of their own) as small weights and of all other instructions as balloons. Then let the rest of the instructions float up and down as they may. The height to which each instruction rises indicates its relative position in the original singleton execution schedule. The relevant formulas are:

$$\text{Ready time back}-\text{propagation} : Ready(n) = Issue(m) - Slack(nm)$$

$$\text{Issue time back}-\text{propagation} : Issue(n) = Ready(n) - ExLat(n)$$

where `m` depends on `n`.

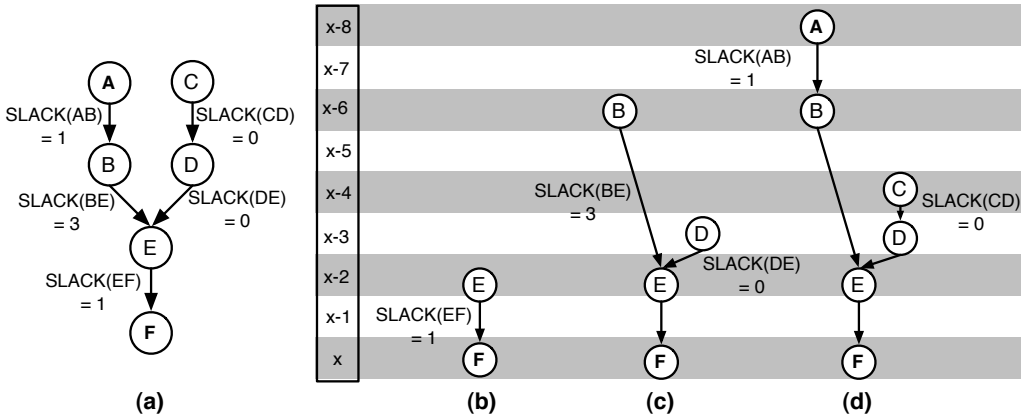


Figure 3.20: Calculating Issue and Ready times from Local Slack Estimates.

Figure 3.20 shows an example of this procedure, beginning with six dependent instructions and the edge local slack values for each dependent edge in Figure 3.20a. Each instruction has an execution latency of 1 cycle. Instruction `F` is the terminal output instruction in the graph, arbitrarily assigned the issue time x. In Figure 3.20b, the issue time of the directly dependent instruction `E` is calculated. `E` has a 1-cycle execution latency and the edge from `E` to `F` has a slack of 1. Applying the ready time propagation formula yields $Ready(E) = Issue(F) - Slack(EF) = x - 1$. Applying the issue time propagation formula we get $Issue(E) = Ready(E) - ExLat(E) = (x - 1) - 1 = x - 2$. In Figure 3.20c, the procedure repeats for instructions `B` and

D along the edges BE and DE respectively. Finally, in Figure 3.20d, the procedure repeats for instructions A and C along edges AB and CD respectively. With this complete execution schedule, it is simple to forward-calculate $Issue$ and $Ready$ times needed for the $\text{Slack}_{\text{Profile}}$ selection algorithm.

Think globally, act locally. In the strictest terms, delay on a given instruction translates into IPC loss only if it consumes more than that instruction's global slack. However, experience shows that local slack is a more useful indicator of mini-graph performance impact than global slack. Indeed, although using global slack yields higher coverage rates (34.8% compared to local slack's 31%), it yields far worse performance (10.9% improvement compared to local slack's 17.6%, with 14% of benchmarks experiencing slowdowns). The problem with global slack is that it is a quantity not exclusive to a single instruction. Whereas local slack is "owned" by a particular instruction, global slack is "shared" by all instructions on a dependence chain to the critical path.

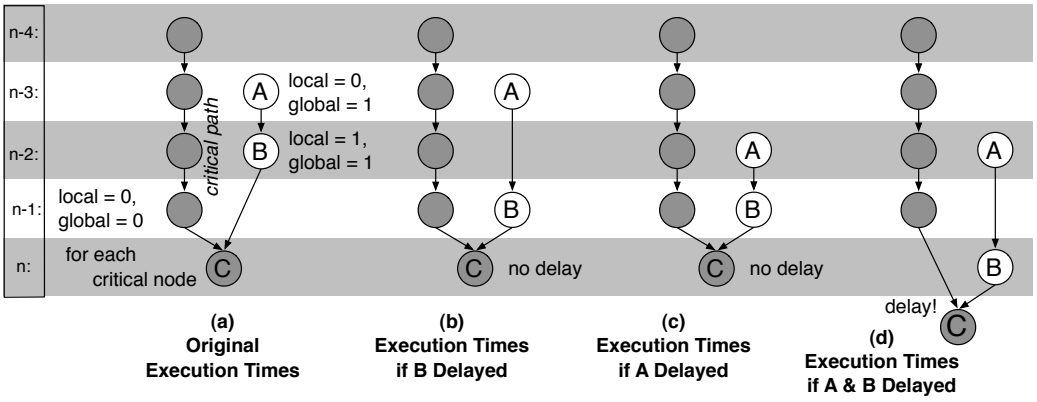


Figure 3.21: Local vs. Global Slack.

Figure 3.21 illustrates. Figure 3.21a shows the execution schedule of a shaded critical path ending in C and two instructions—A and B—on a non-critical path that feeds C. Instruction B is ready one cycle before C needs it, and therefore has 1 cycle of local slack. Because C is a critical instruction, B's 1 cycle of local slack translates

directly into 1 cycle of global slack. (As shown in Figure 3.21b, delaying `B` does not delay the critical instruction `C`.)

Instruction `A`, on the other hand, is ready exactly as instruction `B` needs it, and therefore has no local slack. However, because `B` has 1 cycle of global slack, `A`, too, has 1 cycle of global slack. (As shown in Figure 3.21c, delaying `A` might delay `B`, but it does not delay `C`.)

Selecting Mini-Graphs with Local Slack

```
Identify mini-graphs
Compute local-slack profile on singleton program
Prune mini-graphs
Generate templates
while (template pool not empty & budget not exceeded)
    Choose T, template with largest coverage
    Choose M, all mini-graph instances of T
    Kick out overlaps
```

Selecting Mini-Graphs with Global Slack

```
Identify mini-graphs
Compute global-slack profile on singleton program
Prune mini-graphs
Generate templates
while (template pool not empty & budget not exceeded)
    Choose T, template with largest coverage
    Choose M, all mini-graph instances of T
    Kick out overlaps
    Re-compute global slack profile on new mini-graph program
    Re-prune mini-graphs
```

Figure 3.22: Incorporating Local vs. Global Slack into Selection.

$\text{Slack}_{\text{Profile}}$ decides on a per-mini-graph basis whether a particular delay can be absorbed by the program. If, independently, $\text{Slack}_{\text{Profile}}$ were to permit the consumption of the 1 cycle of global slack held by *both* `A` and `B`, the result is a delay to the critical path (see Figure 3.21d). Avoiding this "over-consumption" of global slack requires a re-profiling pass over the entire program to recalculate the critical path and global slack values for each node. Figure 3.22 illustrates by showing the pseudo code used to incorporate local slack (left) and global slack (right) into the selection algorithm. When using local slack, the profile is obtained once. When using global slack, the profile is re-computed after each mini-graph is selected. This approach is simply not feasible for programs that have potentially tens of thousands of mini-graphs.

The beauty of local slack is that it is owned by one and only one instruction. Tracking local slack allows the delay in Figure 3.21b and *not* the delay in Figure 3.21c. The critical path is not be delayed.

Performance and coverage. Figure 3.23 shows the IPC (left) and coverage

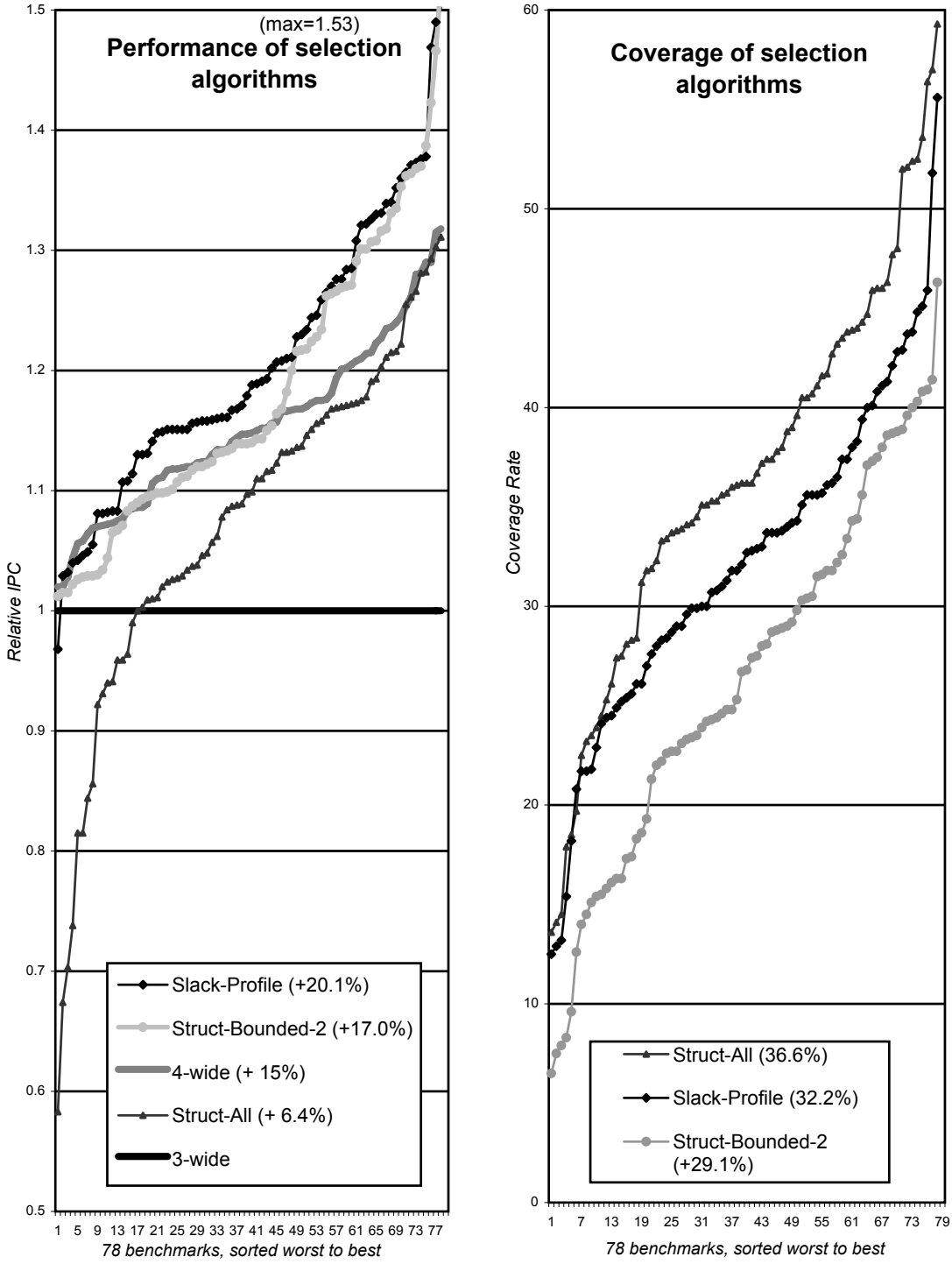


Figure 3.23: Comparison of $\text{Slack}_{\text{Profile}}$ with other models. Left: IPC Relative to a 3-wide processor. Right: Coverage.

(right) of mini-graphs selected using $\text{Struct}_{\text{All}}$, $\text{Struct}_{\text{Bounded}}2$, and $\text{Slack}_{\text{Profile}}$ on a 3-wide processor. $\text{Slack}_{\text{Profile}}$ provides performance that is strictly superior to any other selection scheme. While both $\text{Struct}_{\text{Bounded}}2$ and $\text{Slack}_{\text{Profile}}$ more than match the IPC improvement of actually increasing processor resources to a 4-wide processor, $\text{Slack}_{\text{Profile}}$ outperforms $\text{Struct}_{\text{Bounded}}2$ with an average improvement of 20% (compared to 17%) because of its superior coverage (32% v.s 29%). Only 8 of 78 programs perform worse than than 4-wide processor. A single benchmark, *mcf*, experiences slowdowns with respect to the 3-wide processor. This is due to the fact that $\text{Slack}_{\text{Profile}}$ calculates the execution time of mini-graphs assuming that loads hit in the cache. The poor memory behavior of *mcf* makes this assumption false, and the ability to accurately calculate delays is compromised. The result is a 3% overall slowdown.

The key to $\text{Slack}_{\text{Profile}}$'s success is its combination of aggressive coverage (32% on average) and intelligent serialization avoidance. It is aggressive enough to outperform $\text{Struct}_{\text{All}}$ on the right side of the graph, and selective enough to outperform $\text{Struct}_{\text{None}}$ on the left. $\text{Slack}_{\text{Profile}}$ greatly outperforms the 4-wide "target selector."

Breaking down the model. The $\text{Slack}_{\text{Profile}}$ model has two components: mini-graph-induced instruction delay (rules #1-3) and impact of delay on consumer instructions (rule #4). The graph in Figure 3.24 isolates the contribution of these components. For comparison, the graph also shows full $\text{Slack}_{\text{Profile}}$, $\text{Struct}_{\text{All}}$, and a non-mini-graph 4-wide machine.

Contribution of "consumer delay". $\text{Slack}_{\text{Profile}}$-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 $\text{Slack}_{\text{Profile}}$. On average, explicit accounting for the impact of delay on consumers contributes 2% performance. However, it does change the slope of $\text{Slack}_{\text{Profile}}$, contributing 4% for some programs on the right side of the graph while actually reducing the IPC of

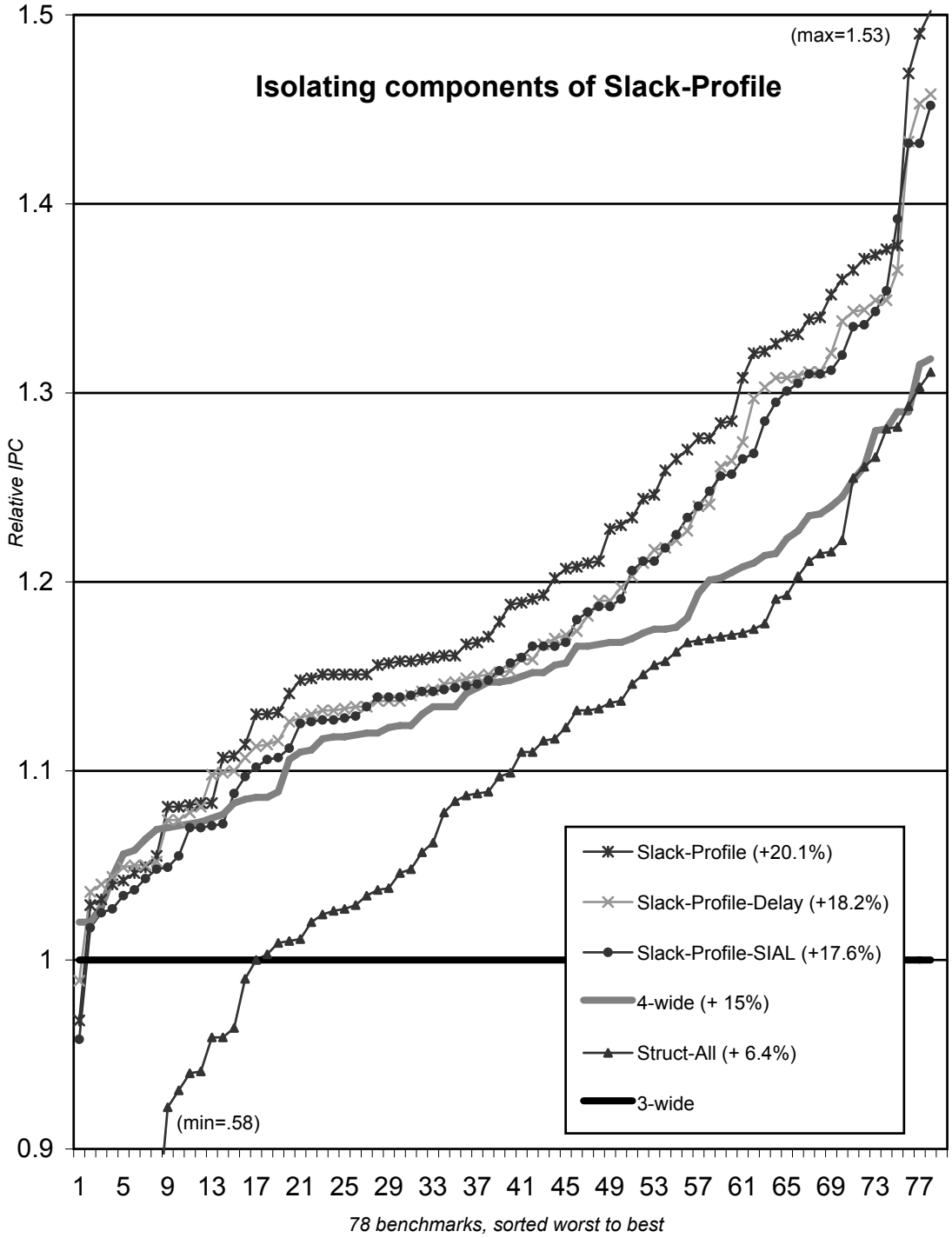


Figure 3.24: Isolating Components of $\text{Slack}_{\text{Profile}}$.

a few programs on the left. Here, slightly reduced coverage is exposing the few pathological mini-graphs that both $\text{Slack}_{\text{Profile}}$ and $\text{Slack}_{\text{Profile}}$-Delay admit.

Contribution of "serialization delay". The difference between $\text{Slack}_{\text{Profile}}$-Delay and $\text{Struct}_{\text{All}}$ corresponds to the contribution of explicit accounting of serialization delay. Obviously, it is this component of the model that accounts for bulk of $\text{Slack}_{\text{Profile}}$'s advantage over serialization-blind selection schemes. $\text{Slack}_{\text{Profile}}$-SIAL (SIAL=Serial Input Arrives Last) is a variant of $\text{Slack}_{\text{Profile}}$-Delay that ignores actual issue delays and focuses only on operand arrival times. Some aggregation schemes [58] use SIAL as their serialization-avoidance heuristic. The difference between $\text{Slack}_{\text{Profile}}$-Delay and $\text{Slack}_{\text{Profile}}$-SIAL shows that explicit accounting for delay is preferred to the operand-arrival-order heuristic. Although the difference is only 0.5% on average, $\text{Slack}_{\text{Profile}}$-SIAL performs a consistent 2% worse than $\text{Slack}_{\text{Profile}}$-Delay on the left-hand side of the graph.

Robustness of slack profiles. In the experiments so far, $\text{Slack}_{\text{Profile}}$—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 graph in Figure 3.25 shows the performance S-curve for all 78 programs running self-trained $\text{Slack}_{\text{Profile}}$ mini-graphs on a 3-wide processor. In addition to

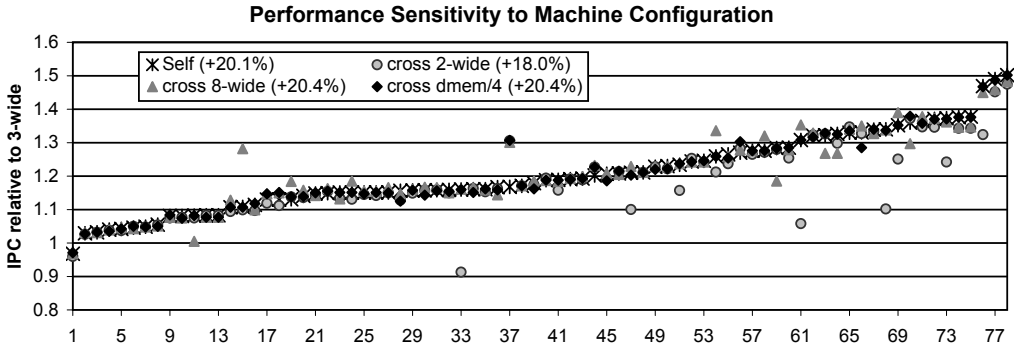


Figure 3.25: $\text{Slack}_{\text{Profile}}$ Robustness: Microarchitecture Sensitivity (Performance). Profiles generated with 2-, 8- wide machines, as well as with a machine with 1/4 the memory hierarchy (cache sizes). Relative Performance. Each vertical set of points compares the three cross-trained with the self-trained (3-wide) results.

this S-curve, the graph also shows results for mini-graphs cross-trained on three configurations: (i) a further reduced 2-wide issue processor (cross 2-wide, circle), (ii) an 8-wide issue processor (cross 8-wide, 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. IPC is occasionally somewhat lower-and coverage somewhat higher-for mini-graphs selected using the 2-wide issue profile (circle). This is intuitive. The 2-wide issue processor generally yields more execution slack as ready instructions wait due to limited issue bandwidth. A profile generated on a relatively reduced machine results 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 machine configuration is detailed further in Figure 3.26. In this

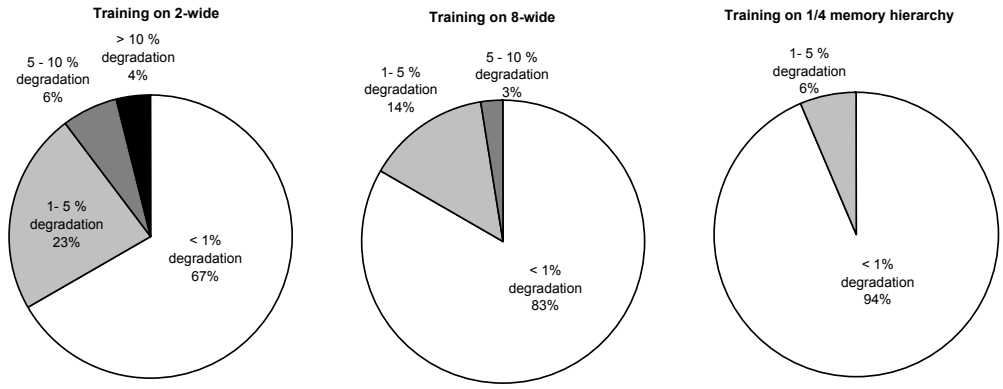


Figure 3.26: $\text{Slack}_{\text{Profile}}$ Robustness: Microarchitecture Sensitivity (Breakdowns). Profiles generated with 2-, 8- wide machines, as well as with a machine with 1/4 the memory hierarchy (cache sizes). Performance Degradation. Breakdown of IPC loss for each machine configuration. Relative to self-trained performance (not 3-wide IPC).

Figure, the IPC degradation associated with each cross training experiment is detailed across all benchmarks. When trained on a 2-wide machine, 90% of benchmarks experience a degradation of less than 5%. When trained on an 8-wide machine or one with a smaller memory hierarchy, this number increases to 97% and 100% of benchmarks. Here, the 5% IPC degradation is relative to the self-trained performance, not the 3-wide IPC. For example, if a benchmark exhibiting a 40% speedup with mini-graphs is degraded by 5%, the IPC improvement is now 33% improvement, not an overall performance loss.

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 left graph of Figure 3.27 shows performance results for SPEC and MiBench programs running $\text{Slack}_{\text{Profile}}$ mini-graphs. In addition to the self-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 SPEC and small for MiBench. The right graph of Figure 3.27 shows a breakdown of how many benchmarks experience slowdowns when

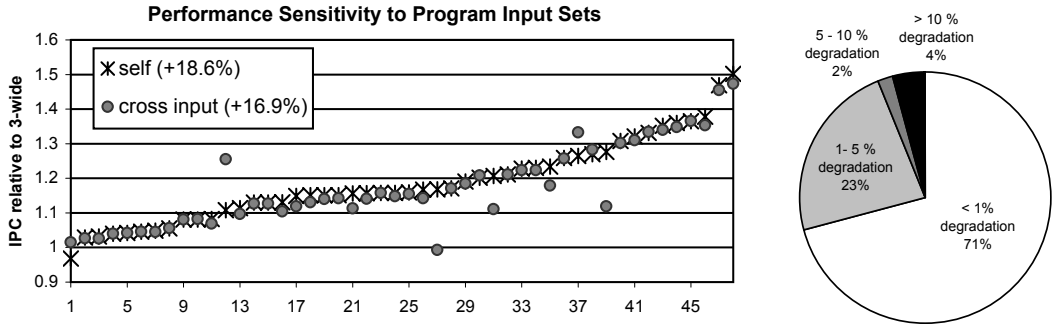


Figure 3.27: $\text{Slack}_{\text{Profile}}$ Robustness: Program Input Sensitivity. Profiles generated with ref inputs. Left: Relative Performance. Each vertical set of points compares the ref-trained with the self-trained results. Right: Performance Degradation. Breakdown of IPC loss relative to self-trained mini-graph runs.

trained on different input sets. 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. Only 1 of 48 benchmarks experiences an IPC *loss* compared to the original 3-wide non-mini-graph performance. 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.

3.4.2 $\text{Slack}_{\text{Dynamic}}$

$\text{Slack}_{\text{Dynamic}}$ implements $\text{Slack}_{\text{Profile}}$ in hardware. Rather than pruning the mini-graph candidate pool before selection, $\text{Slack}_{\text{Dynamic}}$ identifies mini-graphs with harmful serialization at run-time and disables them. As explained in Section 2.2, a mini-graph is disabled by restoring the original outlining jump in the instruction cache and overwriting the mini-graph handle with an actual `nop`. $\text{Slack}_{\text{Dynamic}}$ 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 $\text{Slack}_{\text{Profile}}$, $\text{Slack}_{\text{Dynamic}}$ 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.

Detecting serialization delay. To recognize when a mini-graph is actually delayed by a serializing input, $\text{Slack}_{\text{Dynamic}}$ begins by tracking the last-arriving operands to mini-graphs. This logic, originally introduced as part of the dynamic "shotgun" critical path profiler [33], simply tracks operand arrival order.

Figures 3.28a and 3.28b illustrate. In the first case (Figure 3.28a) the final input arriving to the mini-graph (`r1`) feeds the first instruction of the mini-graph. Although there is a serial input (`r2`) the mini-graph does not wait for this input, and the potential serialization does not manifest. In the second case (3.28b) the final input to the mini-graph (`r2`) *is* the serializing input. This scenario—referred to in this dissertation as *Serial Input Arrives Last* (SIAL)—was adopted as the serialization-avoidance heuristic by macro-op scheduling [58].

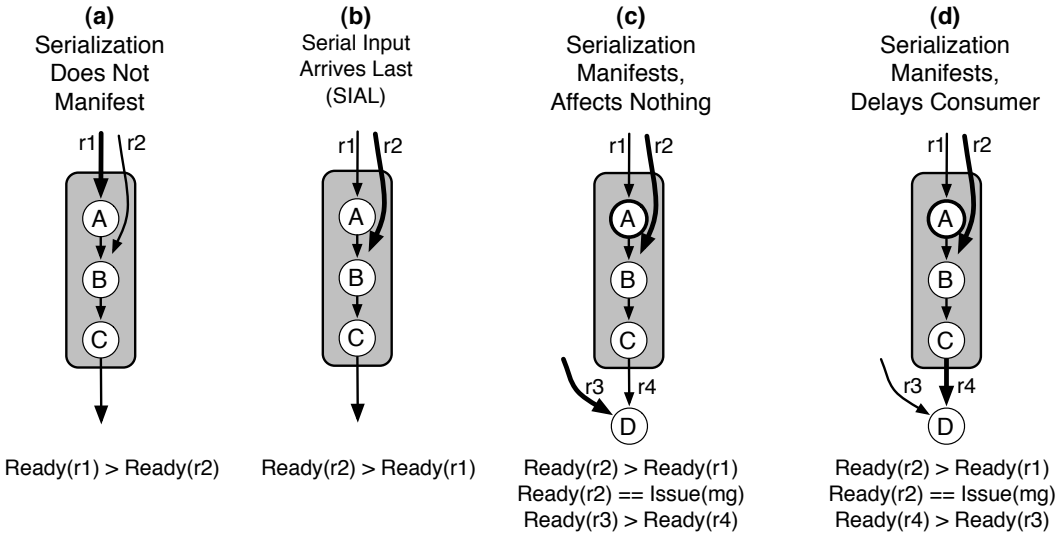


Figure 3.28: Four Dynamic Serialization Scenarios. Bold lines show last incoming edges to an instruction.

One factor that SIAL does not consider is the actual issue time of the mini-graph relative to the arrival of the last operand. $\text{Slack}_{\text{Dynamic}}$ *does* consider this factor. In

$\text{Slack}_{\text{Dynamic}}$, 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.

Detecting consumer delay. To determine whether a mini-graph's serialization delays a consuming instruction, $\text{Slack}_{\text{Dynamic}}$ 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.

Figures 3.28c and 3.28d illustrate. In the first case (Figure 3.28c) the serial input (`r2`) arrives last and the mini-graph issues as this final input arrives, but the mini-graph output (`r4`) is not the last arriving input to the mini-graph consumer `D`. Although serialization appears to have delayed the creation of `r4`, it does not delay the issue time of `r4`'s consumer. In other words, although serialization manifested, it has had no affect. The second case (Figure 3.28d) is identical at first: serialization manifests. This time, however, the mini-graph output, `r4`, *is* the last arriving input to the mini-graph consumer `D`. In this case, serialization not only manifest, but delayed the mini-graph consumer. $\text{Slack}_{\text{Dynamic}}$ disables the mini-graph in Figure 3.28d.

Because execution schedules can change over dynamic instances, $\text{Slack}_{\text{Dynamic}}$ uses a simple saturating counter hysteresis scheme to both avoid rashly disabling a mini-graph that serializes only a few time and to support mini-graph resurrection.

Downsides of dynamic pruning. There are two potential downsides to dynamic mini-graph pruning.

The first is reduced coverage. Static selection algorithms restrict the pool of initial mini-graph candidates; instructions that participate in rejected candidates can still contribute to coverage as participants of other (overlapping) mini-graphs. In contrast, when $\text{Slack}_{\text{Dynamic}}$ rejects a mini-graph, the lost coverage cannot be

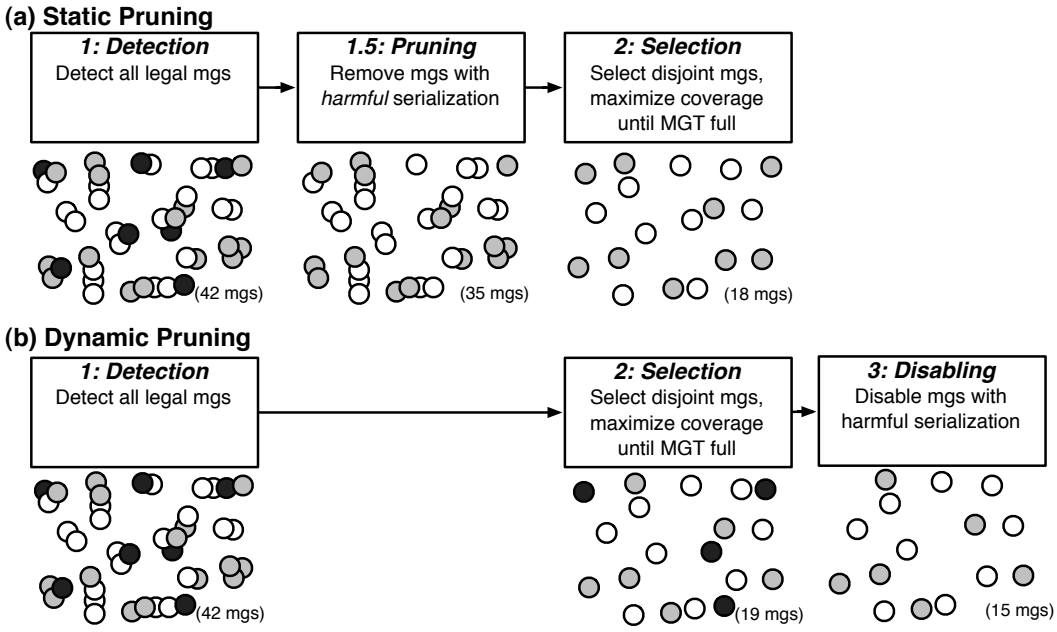


Figure 3.29: Selection with (a) Static Pruning and (b) Dynamic Pruning

reclaimed because the singletons cannot be dynamically re-constituted into new mini-graphs. Dynamic pruning disables mini-graphs with no possibility of replacement.

Figure 3.29 illustrates. Figure 3.29A shows the previously-introduced static pruning algorithm. After pruning the 7 harmful mini-graphs, there are still 35 to choose from. In the case of dynamic pruning, (Figure 3.29B), 4 of the 19 selected mini-graphs are harmful. Once these 4 mini-graphs are disabled, there are only 15 of the original 19 remaining. This downside is common to techniques that combine static aggregation with dynamic pruning.

The second downside is reduced IPC and is a function of mini-graphs' use of outlining, 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; the delay associated with outlining can be worse than the original serialization delay.

Performance and coverage. Figure 3.30 shows the IPC (left) and coverage

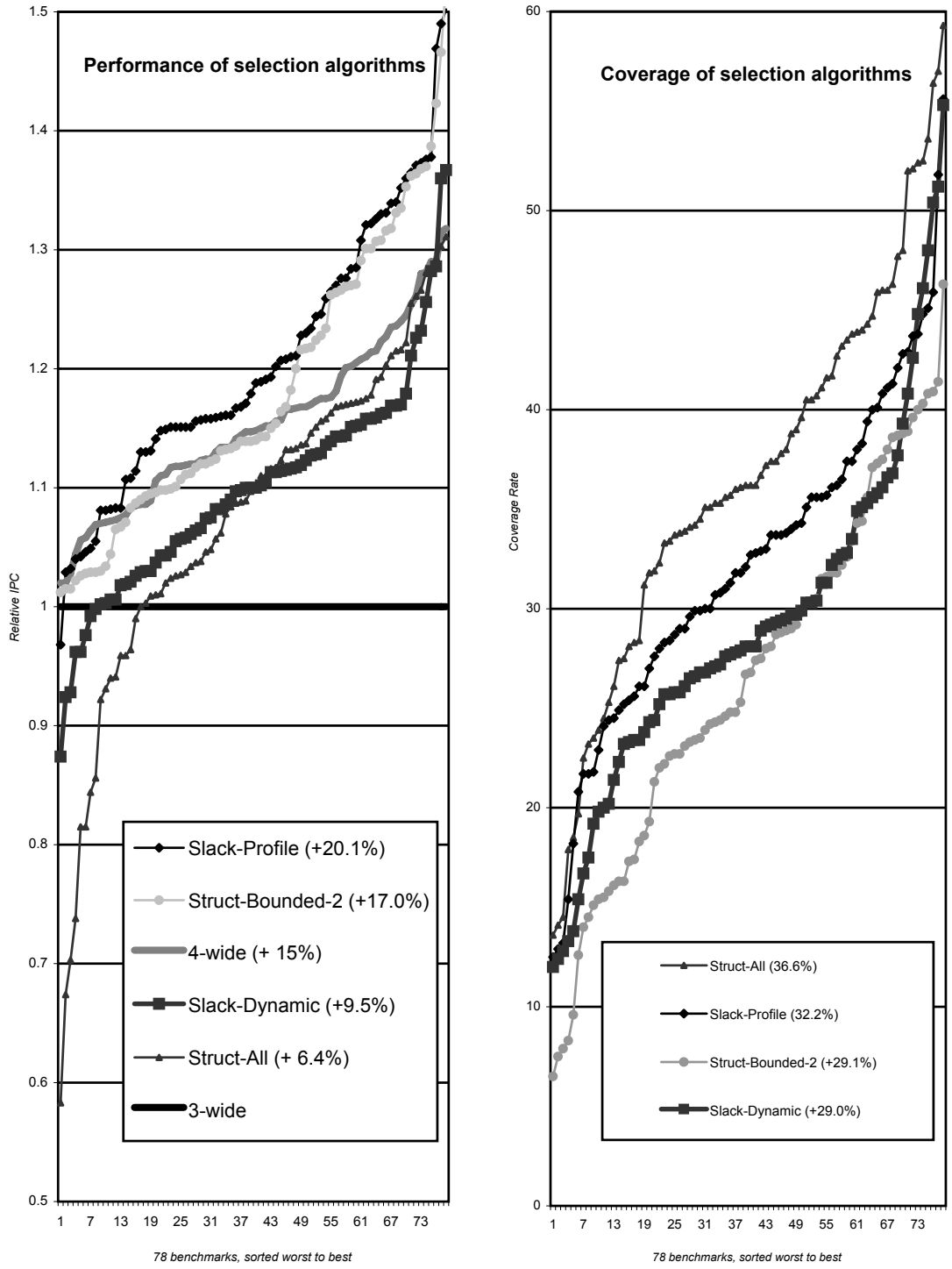


Figure 3.30: Comparison of all Models. Left: IPC Relative to a 3-wide processor. Right: Coverage of algorithms.

(right) of mini-graphs selected using the different slack-based schemes on a 3-wide processor. The graphs include S-curves corresponding to the two structural selectors $\text{Struct}_{\text{All}}$ and $\text{Struct}_{\text{Bounded}}2$. $\text{Slack}_{\text{Dynamic}}$ (box) has similar coverage (29.0%) but worse average IPC improvement (9.5%) than $\text{Struct}_{\text{Bounded}}2$ (29.1% coverage and 17% IPC improvement). $\text{Slack}_{\text{Dynamic}}$ has lower coverage than $\text{Slack}_{\text{Profile}}$ because it cannot re-constitute dynamically-disabled mini-graphs into smaller, more conservative mini-graphs. And although it out-performs $\text{Struct}_{\text{All}}$, $\text{Slack}_{\text{Dynamic}}$ under-performs the other serialization-aware selection schemes. This is because although $\text{Slack}_{\text{Dynamic}}$ eliminates execution serialization penalties, it effectively replaces them with fetch serialization penalties as a disabled mini-graph executes in outlined form, which involves two jumps per instance.

Breaking down the model. Like $\text{Slack}_{\text{Profile}}$, the $\text{Slack}_{\text{Dynamic}}$ serialization avoidance model also has two components: serialization delay and impact on consumers. $\text{Slack}_{\text{Dynamic}}$ 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 3.31 attempts to isolate the other three.

Contribution of outlining penalty. Ideal-$\text{Slack}_{\text{Dynamic}}$ is an implementation of $\text{Slack}_{\text{Dynamic}}$ in which the outlining penalty for disabled mini-graphs is removed. This curve isolates the performance of $\text{Slack}_{\text{Dynamic}}$'s model from the performance effects of the mini-graph encoding scheme. On average, the IPC penalty of outlining degrades $\text{Slack}_{\text{Dynamic}}$'s IPC by 5.5%. Without this penalty, $\text{Slack}_{\text{Dynamic}}$ is much more competitive with $\text{Slack}_{\text{Profile}}$ and almost outperforms a simple 4-wide, non-mini-graph processor.

Contribution of "consumer impact". Ideal-$\text{Slack}_{\text{Dynamic}}$-Delay is a penalty-free $\text{Slack}_{\text{Dynamic}}$ selector that considers only serialization delay, not impact on consuming instructions. This model disables more mini-graphs than $\text{Slack}_{\text{Dynamic}}$. The

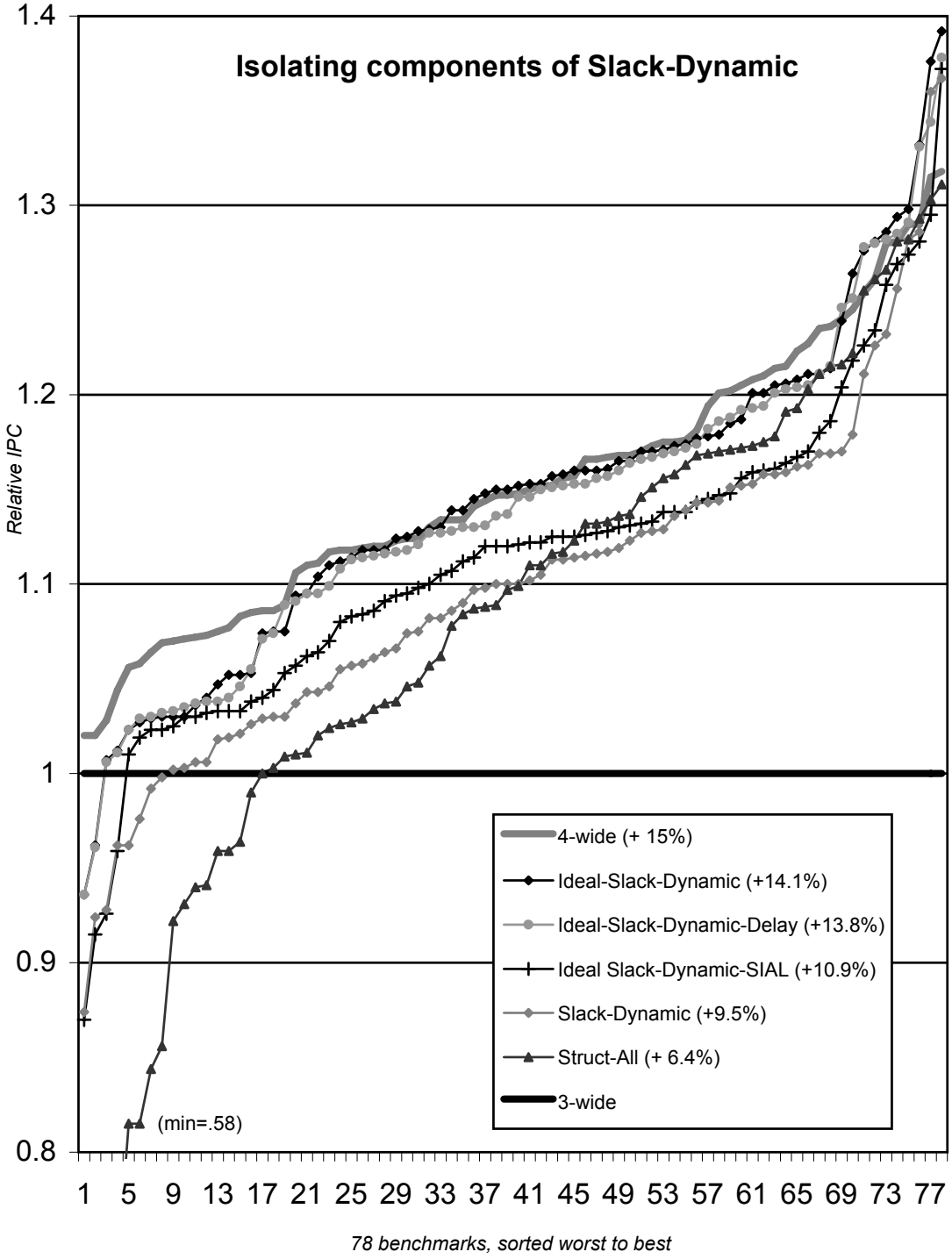


Figure 3.31: Isolating Components of $\text{Slack}_{\text{Dynamic}}$.

contribution of explicit accounting for the potential absorption of delay is the difference between this selector and Ideal-$\text{Slack}_{\text{Dynamic}}$. Interestingly, in the idealized $\text{Slack}_{\text{Dynamic}}$ case, explicit consideration of consumer impact makes less than 1% difference on average. However, when $\text{Slack}_{\text{Dynamic}}$ 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 $\text{Slack}_{\text{Dynamic}}$'s serialization delay detection component is the difference between Ideal-$\text{Slack}_{\text{Dynamic}}$-Delay and $\text{Struct}_{\text{All}}$. As in $\text{Slack}_{\text{Profile}}$, delay accounting provides the bulk of $\text{Slack}_{\text{Dynamic}}$'s impact. Ideal-$\text{Slack}_{\text{Dynamic}}$-SIAL again compares true delay accounting with heuristic tracking of relative operand arrival times. Here—more so than for $\text{Slack}_{\text{Profile}}$—explicit delay accounting provides superior performance, yielding an IPC difference of 3%.

3.4.3 Analysis: Comparison with Exhaustive Search

This section presents a detailed analysis of the selection algorithms algorithms using an exhaustive limit study. Mini-graph selection algorithms choose hundreds of 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 the ten shortest-running benchmarks (*adpcm.encode, adpcm.decode, auto.susan.corners, jpeg.decode, jpeg.encode, mesa.osdemo, office.stringsearch, pegwit.decode, pegwit.encode, unepic*) 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.

For each benchmark, Figure 3.32 shows a coverage (x-axis) and IPC (y-axis) scatter plot for the 1024 possible mini-graph sets running on a 3-wide processor. Again, because only 10 static mini-graphs are considered, this data does not represent the actual coverage or IPC for the 10 benchmarks. The results of each selection algorithm is highlighted at its coverage/IPC intersection.

For most benchmarks, the points on the scatter plot start in the bottom left corner (low coverage, low IPC) and move to the top right corner (high coverage, high IPC). The trend is approximately "coverage improves performance." If this were always the case, the points would simply be in a straight line. The benchmark most closely exhibiting this behavior is *adpcm.decode*, the first—top-left—benchmark in the figure.

However, because some mini-graphs induce performance loss, the points generally form a thick parallelogram rather than single line (*e.g., adpcm.encode, pegwit.encode*). The scatter plot for *unepic* is labeled to show the various possible effects. Beginning with few mini-graphs (no coverage, no IPC change), by adding only helpful mini-graphs (line 1), IPC and coverage increases. By further adding harmful mini-graphs (line 2) coverage continues to increase, but IPC drops. If instead, one were to add only harmful mini-graphs in the beginning (line 3), IPC degrades while coverage increases. Adding helpful mini-graphs (line 4) then increases both coverage and performance again. When different templates affect performance to greater degrees than others, the parallelogram may separate into separate bands (*jpeg.decode, mesa.osdemo*) or even two completely disjoint sets (*office.stringsearch*).

The results for the individual selection algorithms are strikingly intuitive. $\text{Struct}_{\text{All}}$ (medium diamond) occupies the right-most point in each graph; it includes all 10 mini-graphs. $\text{Struct}_{\text{None}}$ (black diamond) occupies the left-most, *i.e.*, lowest coverage, point among all *non-exhaustive* selectors. This may not be the left-most point on the graph (*e.g.*, *adpcm.decode*) because some exhaustive combinations will

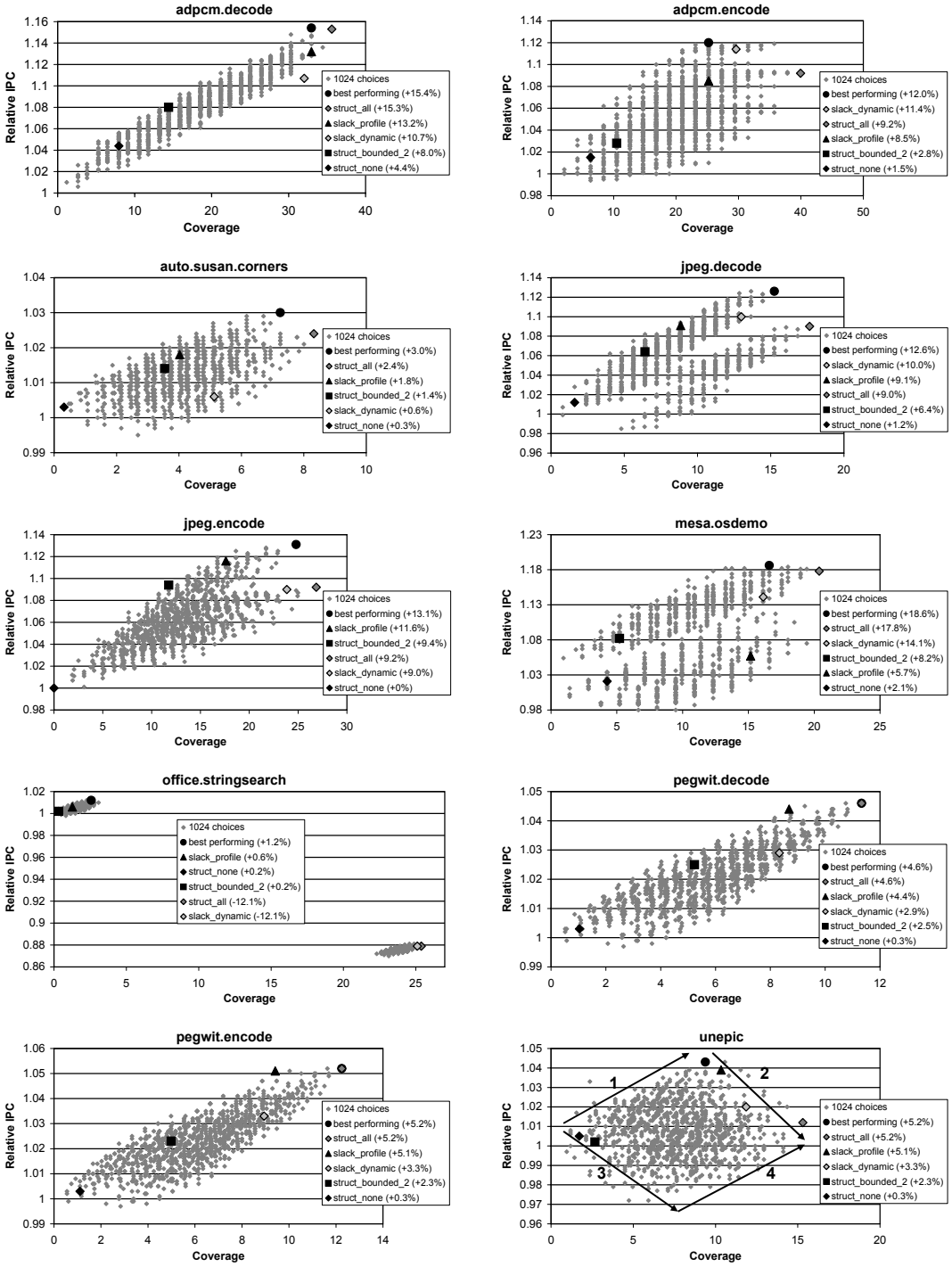


Figure 3.32: Exhaustive Limit Study. Coverage vs. IPC Scatter Plot of all combinations of 10 mini-graph candidates with combinations of five selectors highlighted.

exclude even mini-graphs not subject to serialization, resulting in lower coverage than even $\text{Struct}_{\text{None}}$. $\text{Struct}_{\text{Bounded}}2$ (box), which heuristically allows bounded serialization, typically has higher coverage and IPC than $\text{Struct}_{\text{None}}$, usually still residing in the lower right quadrant of the graph. The quantitative slack-based selectors, $\text{Slack}_{\text{Profile}}$ (triangle) and $\text{Slack}_{\text{Dynamic}}$ (light diamond), combine high coverage with high IPC, with both approaching the IPC of the ideal mini-graph set obtained by exhaustive search.

One nonintuitive result is the position of $\text{Slack}_{\text{Dynamic}}$, especially relative to $\text{Slack}_{\text{Profile}}$. In a realistic selection scenario, $\text{Slack}_{\text{Dynamic}}$ has poorer coverage and IPC than $\text{Slack}_{\text{Profile}}$ 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.

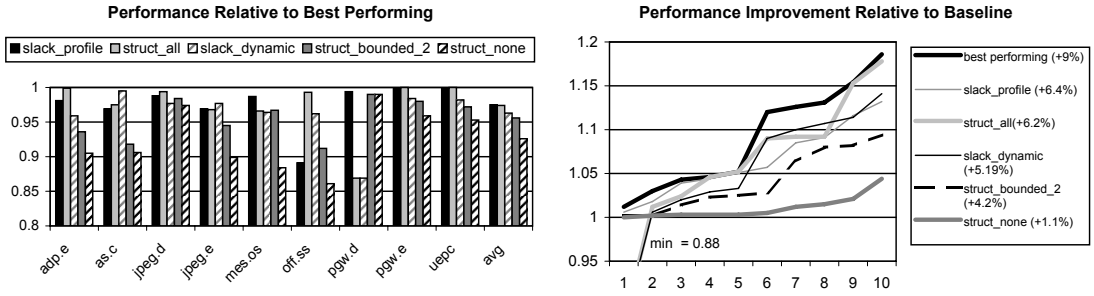


Figure 3.33: Summary of Limit Study. (left) Performance of 5 selectors relative to best performing combination of mini-graphs. (right) S-curve plot of 5 selectors for 10 benchmarks (sorted worst to best).

Figure 3.33 summarizes the individual results of Figure 3.32. On the left, each selector's IPC is shown, relative to the best performing mini-graph combination for that benchmark. The final (11th) set of clusters show average IPC across all 10 benchmarks for each selector. On the right, the familiar s-curve graphs are shown. This graph is of particular interest because—once again—it emphasizes robustness. For example, although *on average* $\text{Struct}_{\text{All}}$ and $\text{Slack}_{\text{Dynamic}}$ perform almost as well

as $\text{Slack}_{\text{Profile}}$, they both induce a 12% slowdown in one case. (And this is with only 10 static mini-graphs!) $\text{Struct}_{\text{Bounded}}2$ and $\text{Struct}_{\text{None}}$ continue to be robust, but unremarkable performance improvements in all cases. $\text{Slack}_{\text{Profile}}$ is both robust and the closest to the best performing (achieving 6.4% improvement compared to the best possible 9%).

Examining $\text{Slack}_{\text{Profile}}$**.** In some cases (*e.g., auto.susan.corners, jpeg.decode*), the best performing set has both higher IPC and higher coverage than $\text{Slack}_{\text{Profile}}$. This means that the best set includes mini-graphs which $\text{Slack}_{\text{Profile}}$ rejects. Although these mini-graphs induce delays by more than their slack, they still improve IPC. There are several possible reasons for this. In some cases, a mini-graph's constituent instructions are so fetch critical that coalescing them into a handle creates enough slack to compensate for the delay. $\text{Slack}_{\text{Profile}}$ does not account for fetch criticality—none of our models do—and thus has no means of assessing such a trade-off. The same effect can occur when ALU resources are the bottleneck; and coalescing instructions into a mini-graph might induce delay but ultimately be beneficial because it increases the number of ALUs available by leveraging the latter stages of the ALU Pipelines.

In some cases (*e.g., unepic*), $\text{Slack}_{\text{Profile}}$ has higher coverage than the best performing set. In this case, $\text{Slack}_{\text{Profile}}$ includes a mini-graph which the best performing set excludes. When chosen in isolation, this mini-graph actually improves IPC! That this mini-graph is not included in the best performing set speaks to the fact that mini-graph selection is non-decomposable. (Similarly, sometimes mini-graphs without any serialization vulnerability are 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 $\text{Slack}_{\text{Profile}}$ does not re-profile local slack after each mini-graph is selected, it assesses mini-graphs in isolation without considering potential interference.

3.5 Related Work on Aggregate Selection Schemes

The selection algorithms presented in this chapter address the problem of serialization. Although these algorithms represent the most extensive treatment of the subject of serialization, they are not the sole attempts at avoiding serialization penalties. Furthermore, although the algorithms here are presented and evaluated in the context of mini-graphs, their applicability should extend to other aggregate schemes [23, 58, 89, 96].

3.5.1 Treatment of Serialization

A few (non-mini-graph) aggregation proposals address the subject of serialization. Macro-op scheduling [58] avoids "harmful grouping" by disabling macro-ops whose serializing input is the last-arriving operand. This approach is referred to in Section 3.4.1 as *Serial Input Arrives Last* (SIAL). In contrast, $\text{Slack}_{\text{Dynamic}}$ detects actual issue delay and the impact it has on dependent instructions.

Macro-op execution [47] 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 $\text{Struct}_{\text{Bounded}}2$, 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.

CCA graph selection [21] 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.

3.5.2 Applicability to other Aggregate Schemes

Although presented in the context of mini-graphs, the selection algorithms presented in this chapter should be applicable to the other aggregation schemes that target dynamically scheduled processors [21, 58, 89]. Different aggregate schemes offer different benefits. Benefits are a function of 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.

The selection algorithms presented in this chapter are applicable to other aggregation schemes (in varying degrees) because they focus on the aspect of aggregation that is common to all of them: the cost of serialization. Although as presented, $\text{Struct}_{\text{Bounded}}2$ and $\text{Slack}_{\text{Profile}}$ explicitly model internal serialization, this aspect of the model can be easily removed if it does not apply. $\text{Slack}_{\text{Dynamic}}$ only implicitly accounts for internal serialization.

The presented 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 [21, 89] or explicit tolerance of some latency in the underlying micro-architecture, *e.g.*, a pipelined scheduler [58].

Early experiments with macro-op scheduling indicate that $\text{Slack}_{\text{Profile}}$ (modulo the penalties of outlining and reduced coverage) is a more selective, better performing serialization-filter than macro-op's own SIAL heuristic and that a re-targeted version of $\text{Slack}_{\text{Profile}}$ produces better results than the heuristics used by macro-op execution

[47]. The SIAL results presented in Section 3.4.1 provide additional support for this observation, albeit in the context of mini-graphs, not macro-ops.

3.6 Summary

Mini-graph processing extends the benefits of instruction fusion to more pipeline stages, processor structures, and dynamic instructions than most other instruction fusion techniques. The benefit of this is increased resource amplification. The cost is serialization. Serialization, the introduction of an artificial dependence between two instructions, is specifically introduced by virtue of issuing and executing instruction sequences—particularly, but not exclusively, independent sequences—as an atomic unit. Serialization can delay the execution of instructions found within a mini-graph, causing performance degradation in some benchmarks.

This chapter introduces techniques for avoiding serialization while still maximizing coverage. The challenge lies in the fact that structural inspection of a mini-graph is not sufficient for detecting mini-graphs that actually lead to performance degradations. Some mini-graphs that are vulnerable to serialization do not actually induce execution delays in practice. Furthermore, some execution delays do not impact total program performance. A slack-based approach to detecting harmful serialization is able to detect not only which instructions will induce execution delays when placed in a mini-graph, but which execution delays will actually affect the overall performance of a program. By detecting and pruning out the rare, pathological cases of performance-degrading mini-graphs, the techniques introduced in this chapter manage to remove the performance threat of serialization while maintaining the amplification benefits of mini-graph processing.

Chapter 4

Performance Analysis

This chapter explores the performance benefits of mini-graph processing. Section 4.1 shows a characterization of the four benchmarks suites used, showing how various functional characteristics lead to more or less mini-graph coverage and how various characteristics allow that coverage to translate to performance improvement. Section 4.2 analyzes the contributions of the different amplification benefits of mini-graph processing to performance improvement. Section 4.3 analyzes mini-graph performance improvement on an in-order machine and shows how performance contributors change in this new setting. Because Sections 4.2 and 4.3 both show instruction cache capacity amplification to have a limited role in performance improvement, Section 4.4 shows how instruction cache capacity amplification plays a greater role in performance improvement with smaller instruction cache sizes. Section 4.5 considers the coverage and performance implications of various MGT configurations.

4.1 Benchmark Suites

Mini-graph binaries are created from 78 Alpha binaries across four benchmark suites: SPECint2000 (SPEC), MediaBench [60], CommBench [108], and MiBench [42]. The original binaries are compiled for the Alpha EV6 using the Digital OSF compiler with

optimization flags -O3. All benchmarks are 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. They are included to show the applicability of mini-graphs to different kinds of codes.

Tables 4.1 through 4.4 show a characterization of the 78 programs, with both suite and total averages. Tables 4.2 and 4.2 show functional properties of the benchmarks, beginning with the instruction mix. The first four classes are instructions that can participate in integer-memory mini-graphs: loads, stores, conditional branches, and single-cycle ALU operations. Single-cycle ALU operations are significant because these instructions can occur multiple times per mini-graph, unlike memory instructions and conditional branches. The SPEC suite—with more memory instructions and fewer single-cycle ALU operations—on the one hand sees the greatest increased coverage with integer-memory mini-graphs and on the other hand is the most limited in forming mini-graphs in the first place. The percentage of floating point operations is shown to further characterize the benchmarks.

Next, the average dynamic basic block size is shown, where each basic block's size is weighted by execution frequency. Smaller basic blocks translate to fewer and smaller mini-graph, because mini-graphs do not span basic block boundaries. Of the four benchmark suites, SPEC has smallest basic blocks. Finally, the integer and integer-memory mini-graph coverage rates for each benchmark are shown.

The microarchitectural characterization, shown in Tables 4.3 and 4.4, begins with the IPC of 3-wide and 4-wide processors, both with and without mini-graph processing as well as mini-graph–to–baseline IPC ratios (*MG/3w*, *MG/4w*) and the 4-wide–to–3-wide IPC ratio (*4w/3w*). As expected, the amplification offered by mini-graph processing yields more performance improvement on the 3-wide machine than it does on the more-provisioned 4-wide machine. Total and individual suite results are shown at the bottom of Table 4.4. Once again, on average, a 3-wide mini-graph

suite	benchmark	instruction mix %					dynamic basic block size	int coverage (%)	intmem coverage (%)
		load	store	control	1c ALU	FP			
SPEC	bzip2	25.5	8.1	13	50.3	0	7.1	7.6	32.9
	crafty	28.2	5.4	12.1	51.5	0	7.2	13.3	34.2
	eon.cook	24.7	19.5	11.4	25	14.3	9.7	4.4	13.2
	eon.kajiya	25.2	21.2	11.6	23	13.9	9.5	3.3	12.5
	eon.rushmeier	23.6	22.1	12	24.4	12.4	9.5	3.6	12.9
	gap	24.7	9.6	15.7	44.7	0.1	5.7	9.3	29
	gcc	23.5	15.6	16.9	38.6	0	4.5	3.5	21.7
	gzip	20.2	6.7	12.2	59.7	0	9.8	18.2	38.3
	mcf	28.8	6.1	21.8	30.6	0	4.5	4.8	33
	parser	21.7	8.9	17.1	46.1	0.1	4.7	13.9	36.1
	perlbmk.diffmail	27.8	16.8	14.9	36.6	0.1	6.4	5.5	24.1
	perlbmk.scrabbl	30	15.2	15	34.4	0.4	5.9	5.7	24.9
	twolf	22.3	7.2	13.4	49.4	4.5	5.2	11.6	29.9
	vortex	27	16.9	17.8	32.1	0.2	5.4	2.6	25.4
	vpr.place	26.3	8.6	11.6	44.6	6.7	6.2	5.5	20.8
	vpr.route	30.5	11.7	11.5	35.2	7	9.1	4.9	15.4
MEDIABENCH	adpcm.dec	6.6	2.7	10.6	77.4	0	3.8	28.7	34
	adpcm.enc	6.3	1.1	11.6	78.9	0	4.9	22.1	40
	epic	13.4	2.3	13.8	47.9	21.2	5.1	2.3	25.2
	g721.dec	12.4	3.7	19	59.8	1.4	4.7	16.8	42.9
	g721.enc	12	3.5	19	60.4	1.4	4.9	17.2	42.8
	gs.dec	18.9	12	16.6	46.2	0.4	6.2	9.2	28.7
	gsm.dec	6.7	3.8	8.5	71.7	2.7	8.7	25.3	40.8
	gsm.enc	15.6	2.3	3.4	68.3	1.3	15.7	31.1	39.4
	jpeg.dec	19.1	8.6	5.6	62.6	0	18.2	19.9	42.1
	jpeg.enc	21.7	7.3	12.8	56.4	0	4.9	14.6	45.9
	mesa.mipmap	26.2	12.8	8	35.8	9.4	8.8	7.8	26.1
	mesa.osdemo	28.8	14.3	8	38.1	1.5	9.1	9.9	33.8
	mesa.texgen	23.7	11.5	7.6	38.1	10.1	9.2	5.9	21.7
	mpeg2.dec	16.5	6.1	8.8	51.9	15.9	5.2	9.7	30
	mpeg2.enc	26.1	2.9	5.7	59.9	3.6	11.7	25.1	34.3
	pegwit.dec	20.3	5.6	10.5	61.2	1.5	6.9	12.6	35.6
	pegwit.enc	18.1	5	10.8	63.6	1.5	5.5	14.2	36.2
	unepic	12.6	9.4	16.1	54.2	4.5	6.2	10.5	32.7
COMM	cast.dec	21.6	9.4	6.8	56.3	1.5	19.2	17.6	35.6
	cast.enc	21.6	9.4	6.8	56.3	1.5	19.2	17.7	35.6
	drr	35.5	9.4	19.9	20.4	4.6	3.6	1.1	33.7
	frag	13	7.1	16.9	53.6	6.8	5.5	7.6	33.7
	gzip.dec	19.1	4	12.9	61.7	0	6.6	12	43.8
	gzip.enc	16.9	4.7	14.2	62.5	0	6.4	16.4	55.6
	jpeg.dec	18.4	9.2	6.8	60.3	0.5	11.5	18.4	37.4
	jpeg.enc	18.7	11.3	10.9	53.9	0.5	5	10.8	38
	reed.dec	12.2	5.7	18.1	60.1	0.5	4.4	10.1	45.1
	reed.enc	13.5	4.5	15.2	54.2	8.5	4.5	10.4	35.1
	rtr	25.5	6.8	18.8	44.1	0.4	6.3	6.3	27.6
	tcpdump	18.3	11	18.7	46.1	0.2	5	8.4	30.7

Table 4.1: Functional Properties of Benchmarks across 3 suites: SPEC, MediaBench, and CommBench

suite		benchmark	instruction mix % load	store	control	1c ALU	FP	dynamic basic block size	int coverage (%)	intmem coverage (%)
MI Bench	*Auto*	basicmath	15.1	9.7	16.4	53.3	1.5	5.7	9.1	31.8
		bitcount	7.8	2.4	14.8	71.4	0.4	4.7	17.1	36.5
		qsort	21.2	9.6	14.7	48.9	2	5.1	8.1	31
		susan (corners)	32.7	1.2	6.5	57.2	0.7	11.5	7.6	27
		susan (edges)	29.2	1.2	5.9	55.6	4	12	7	29.6
		susan (smoothing)	22.2	0.3	8.6	54.5	0	4.5	0.8	24.5
	Consumer	jpeg.dec	21	10.4	5.5	59.3	0	23.1	16.3	41.1
		jpeg.enc	22.3	7.3	11.6	57.1	0	5.2	15.4	44.8
		lame	24.7	8.7	6.5	29.8	28.8	11	7.8	18.2
		tiff (2bw)	18.8	8.1	5.9	44.9	0	10.2	5.5	22.9
		tiff (2rgba)	25	14.2	2.2	55.1	0	2.6	14.7	24.4
		tiff (dither)	14	6.8	14.5	62.9	0	4.7	24.4	43.7
		tiff (median)	24.9	4.8	12.9	55.1	0	1.8	10.6	51.8
	Net	dijkstra	21.3	6.8	14.6	45.9	2.5	5.9	8.1	33.7
		patricia	22.2	9.8	17.2	46.3	0.3	5.2	8.4	29
	Office	ghostscript	19.2	12.4	16.4	46.3	0.1	6.3	10.1	30.8
		ispell	19.3	11.7	16.8	45.6	0.8	4.6	8.2	30
		rsynth	25.5	8.7	10.6	30.1	24.4	7.5	9	21.8
		stringsearch	10	14.1	19.7	52.8	0.3	5	4.2	41.3
	Security	blowfish.dec	22.3	10.7	7.4	51.4	1	6.3	13	32.8
		blowfish.enc	22.3	10.7	7.4	51.5	1	6.5	13.2	32.1
		pgp	19.2	6.9	8.5	58.5	0	7.4	9	31.8
		rijndael.dec	28.9	7.1	5.8	53.8	1.7	13.4	15.7	35.7
		rijndael.enc	29.7	7	5.4	55.1	0.4	13.7	16.3	37.4
		sha	13.5	3.6	5.9	64.3	0.8	5.4	16.8	28.4
	Telecom	ADPCM.dec	6.4	2.6	10.3	78.2	0	3.9	12.8	25.6
		ADPCM.enc	6.1	1	11.1	79.8	0	4.4	14.1	31.3
		CRC32	21.7	13	13	47.8	0	4.5	0	26.1
		FFT	16.1	9	11	49.3	11.7	8.9	12.8	28
		IFFT	15.8	8.9	11.3	50.3	10.8	8.5	12.2	28.3
		GSM.dec	8.3	4.6	8.8	69.6	2	8.8	22.6	40.1
		GSM.enc	15.7	2.4	3.5	67.1	2.2	16.3	22.2	29.9
SPEC			**25.6**	**12.5**	**14.3**	**39.1**	**3.7**	**6.9**	**7.4**	**25.3**
MEDIABENCH			**16.9**	**6.4**	**10.9**	**57.4**	**4.2**	**7.8**	**15.7**	**35.1**
COMM			**19.5**	**7.7**	**13.8**	**52.5**	**2.1**	**8.1**	**11.4**	**37.7**
MI BENCH			**19.5**	**7.4**	**10.3**	**54.7**	**3**	**7.6**	**11.7**	**31.9**
ALL 78			**20.2**	**8.2**	**11.8**	**51.8**	**3.3**	**7.6**	**11.7**	**32.2**

Table 4.2: Functional Properties of Benchmarks in MiBench, averages of each suite, and all 78 programs.

suite	benchmark	IPC							I$ miss rate (%)	D$ miss rate (%)	mini-graph supported bookkeeping reductions (%		
		3-wide			4wide						% insns fetched	% RS entries	% Reg. Writes
		base	MG	**MG/3w**	base	**4w/3w**	MG	MG/4w					
SPEC	bzip2	1.59	1.89	**1.19**	1.87	**1.18**	2.01	1.07	0.00	2.42	29.5	30.5	26.5
	crafty	1.91	2.30	**1.20**	2.20	**1.16**	2.51	1.14	0.13	1.67	25.9	27.9	25.3
	eon.cook	2.08	2.20	**1.05**	2.26	**1.08**	2.34	1.04	0.01	0.56	9.4	10.0	8.8
	eon.kajiya	1.93	2.00	**1.04**	2.06	**1.07**	2.11	1.03	0.03	0.36	7.7	8.6	7.2
	eon.rushmeier	2.02	2.08	**1.03**	2.17	**1.07**	2.20	1.01	0.02	0.43	8.6	9.3	7.7
	gap	0.94	0.98	**1.04**	1.00	**1.06**	1.02	1.03	0.03	2.98	21.6	23.1	20.0
	gcc	1.38	1.48	**1.07**	1.48	**1.08**	1.56	1.05	0.48	5.05	14.3	15.6	12.0
	gzip	1.81	2.07	**1.14**	2.03	**1.12**	2.19	1.08	0.00	4.04	28.0	29.7	27.0
	mcf	0.21	0.20	**0.96**	0.22	**1.06**	0.21	0.98	0.00	32.90	24.9	27.5	19.8
	parser	1.00	1.03	**1.03**	1.07	**1.07**	1.08	1.01	0.00	4.43	26.0	28.3	22.5
	perlbmk.diffmail	1.50	1.66	**1.11**	1.63	**1.09**	1.75	1.07	0.37	1.21	12.6	14.0	9.9
	perlbmk.scrabbl	1.61	1.83	**1.14**	1.75	**1.09**	1.97	1.13	0.41	0.67	17.0	19.0	15.2
	twolf	1.38	1.59	**1.15**	1.53	**1.11**	1.68	1.10	0.00	9.06	23.9	26.3	23.1
	vortex	1.99	2.29	**1.15**	2.30	**1.16**	2.54	1.11	0.39	0.81	19.4	19.8	13.3
	vpr.place	1.55	1.79	**1.15**	1.81	**1.17**	2.00	1.10	0.00	4.70	15.5	17.1	14.9
	vpr.route	0.93	0.96	**1.03**	1.01	**1.09**	1.05	1.04	0.01	5.01	11.3	12.3	9.0
MEDIABENCH	adpcm.dec	1.68	1.94	**1.15**	1.88	**1.12**	2.06	1.10	0.00	0.04	17.3	19.1	22.4
	adpcm.enc	1.41	1.56	**1.10**	1.52	**1.07**	1.56	1.03	0.00	0.04	22.5	27.0	28.3
	epic	2.36	2.73	**1.16**	2.72	**1.15**	3.31	1.22	0.00	1.94	21.4	22.4	15.2
	g721.dec	2.16	2.79	**1.30**	2.52	**1.17**	2.93	1.17	0.00	0.00	33.3	35.7	33.3
	g721.enc	2.01	2.55	**1.27**	2.32	**1.15**	2.73	1.18	0.00	0.00	30.8	33.5	32.1
	gs.dec	2.28	2.73	**1.20**	2.67	**1.17**	2.95	1.10	0.02	0.18	23.7	25.0	18.8
	gsm.dec	2.43	3.21	**1.32**	3.13	**1.29**	3.85	1.23	0.00	0.01	37.9	39.3	35.8
	gsm.enc	2.57	3.77	**1.47**	3.39	**1.32**	4.54	1.34	0.00	0.00	36.3	36.9	36.4
	jpeg.dec	2.04	2.59	**1.27**	2.49	**1.22**	2.79	1.12	0.04	0.52	36.4	37.9	39.5
	jpeg.enc	2.11	2.72	**1.29**	2.48	**1.17**	2.87	1.15	0.01	0.74	39.5	40.8	37.6
	mesa.mipmap	2.10	2.56	**1.22**	2.44	**1.17**	2.92	1.19	0.00	0.48	23.7	23.7	22.2
	mesa.osdemo	2.30	3.00	**1.31**	2.82	**1.23**	3.54	1.26	0.01	0.62	31.1	31.5	28.4
	mesa.texgen	2.19	2.58	**1.18**	2.64	**1.20**	3.03	1.15	0.02	0.34	18.9	19.3	17.8
	mpeg2.dec	2.57	3.26	**1.27**	3.19	**1.24**	3.63	1.14	0.00	0.15	27.6	28.3	23.4
	mpeg2.enc	2.56	3.38	**1.32**	3.16	**1.24**	3.81	1.20	0.00	0.10	31.5	32.4	30.3
	pegwit.dec	1.71	1.92	**1.12**	2.00	**1.17**	2.09	1.05	0.01	8.08	21.8	24.0	20.8
	pegwit.enc	1.71	1.94	**1.14**	2.00	**1.17**	2.12	1.06	0.00	9.09	22.2	24.4	21.3
	unepic	2.00	2.34	**1.17**	2.23	**1.12**	2.51	1.12	0.01	3.65	27.0	27.8	21.5
COMM	cast.dec	2.13	2.25	**1.05**	2.18	**1.02**	2.42	1.11	0.00	0.02	32.6	33.1	30.5
	cast.enc	2.13	2.25	**1.05**	2.18	**1.02**	2.42	1.11	0.00	0.02	32.6	33.1	30.5
	drr	2.21	2.58	**1.16**	2.51	**1.13**	2.60	1.04	0.00	0.89	30.0	29.3	21.1
	frag	2.28	2.52	**1.11**	2.38	**1.04**	2.65	1.12	0.00	0.07	32.4	33.8	31.4
	gzip.dec	1.90	2.09	**1.10**	2.14	**1.12**	2.12	0.99	0.00	0.90	35.4	37.6	33.2
	gzip.enc	1.57	1.83	**1.17**	1.68	**1.07**	1.88	1.12	0.00	8.90	42.8	46.1	43.6
	jpeg.dec	2.53	3.25	**1.29**	3.07	**1.22**	3.59	1.17	0.00	0.85	34.1	35.2	34.8
	jpeg.enc	2.17	2.76	**1.27**	2.55	**1.18**	2.97	1.16	0.00	0.64	32.1	33.6	27.9
	reed.dec	2.20	2.78	**1.27**	2.65	**1.21**	2.96	1.12	0.00	0.00	36.6	39.5	27.5
	reed.enc	1.88	2.30	**1.22**	2.12	**1.12**	2.57	1.22	0.00	0.00	21.7	24.7	19.2
	rtr	1.97	2.24	**1.14**	2.21	**1.12**	2.47	1.12	0.00	0.14	24.9	25.4	19.0
	tcpdump	1.97	2.31	**1.18**	2.20	**1.12**	2.46	1.12	0.05	0.08	21.3	22.9	16.7

Table 4.3: Microarchitectural Properties of Benchmarks across 3 suites: SPEC, MediaBench, and CommBench.

suite		benchmark	IPC							I$ miss rate (%)	D$ miss rate (%)	mini-graph supported bookkeeping reductions (%		
			3-wide			4wide						% insns fetched	% RS entries	% Reg. Writes
			base	MG	**MG/3w**	base	**4w/3w**	MG	MG/4w					
MI Bench	*Auto*	basicmath	2.25	2.63	**1.17**	2.58	**1.15**	2.88	1.12	0.00	0.00	25.8	27.1	19.9
		bitcount	2.42	3.07	**1.27**	2.76	**1.14**	3.16	1.15	0.00	0.00	33.6	32.9	29.9
		qsort	2.43	2.79	**1.15**	2.78	**1.14**	3.33	1.20	0.00	0.19	26.5	27.3	19.2
		susan (corners)	2.16	2.50	**1.15**	2.60	**1.20**	2.79	1.07	0.00	0.66	23.3	24.4	20.9
		susan (edges)	2.19	2.63	**1.20**	2.55	**1.17**	2.87	1.12	0.00	0.48	26.1	27.0	25.2
		susan (smoothing)	2.53	2.95	**1.17**	2.83	**1.12**	3.62	1.28	0.00	0.03	20.8	22.5	16.2
	Consumer	jpeg.dec	2.30	2.99	**1.30**	2.94	**1.28**	3.29	1.12	0.01	0.41	36.3	37.3	37.6
		jpeg.enc	2.25	2.97	**1.32**	2.73	**1.21**	3.20	1.17	0.00	0.72	39.7	40.8	37.7
		lame	2.03	2.19	**1.08**	2.21	**1.09**	2.33	1.05	0.08	2.09	15.3	15.9	14.3
		tiff (2bw)	2.73	3.26	**1.19**	3.29	**1.20**	4.00	1.22	0.00	1.11	22.1	22.1	12.2
		tiff (2rgba)	1.73	1.93	**1.12**	2.06	**1.19**	2.17	1.05	0.00	6.31	23.7	23.8	25.6
		tiff (dither)	2.26	2.85	**1.26**	2.56	**1.13**	3.07	1.20	0.00	0.30	34.5	37.3	38.6
		tiff (median)	2.65	3.40	**1.28**	3.31	**1.25**	3.55	1.07	0.00	2.13	50.3	50.7	48.6
	Net	dijkstra	2.07	2.59	**1.26**	2.55	**1.23**	2.89	1.14	0.00	0.64	32.1	32.5	20.8
		patricia	1.55	1.94	**1.25**	1.67	**1.08**	2.15	1.29	1.32	0.61	23.0	24.2	18.6
	Office	ghostscript	2.23	2.73	**1.22**	2.62	**1.17**	3.04	1.16	0.02	0.19	25.4	26.8	21.2
		ispell	1.92	2.20	**1.15**	2.20	**1.15**	2.47	1.12	0.01	1.36	23.7	24.9	17.4
		rsynth	2.33	2.44	**1.05**	2.39	**1.03**	2.61	1.09	0.00	0.04	20.5	21.2	20.2
		stringsearch	1.95	2.10	**1.08**	2.08	**1.06**	2.18	1.05	0.01	0.23	24.8	27.0	19.5
	Security	blowfish.dec	2.38	2.69	**1.13**	2.65	**1.11**	2.87	1.08	0.00	0.01	28.6	29.1	25.8
		blowfish.enc	2.38	2.69	**1.13**	2.65	**1.11**	2.88	1.09	0.00	0.01	28.6	29.0	25.8
		pgp	2.58	3.34	**1.30**	3.23	**1.25**	3.94	1.22	0.01	0.03	28.7	29.8	19.6
		rijndael.dec	2.81	3.65	**1.30**	3.55	**1.26**	4.02	1.13	0.00	0.45	33.2	33.5	30.0
		rijndael.enc	2.77	3.66	**1.32**	3.54	**1.28**	4.12	1.16	0.00	0.87	35.2	35.7	33.9
		sha	2.50	3.10	**1.24**	2.84	**1.13**	3.50	1.23	0.00	0.00	25.2	26.1	22.3
	Telecom	ADPCM.dec	1.84	1.99	**1.08**	2.08	**1.13**	2.21	1.06	0.00	0.00	9.7	11.5	12.1
		ADPCM.enc	1.69	1.89	**1.12**	1.89	**1.12**	2.01	1.06	0.00	0.00	20.9	24.0	24.9
		CRC32	2.70	3.28	**1.21**	3.28	**1.21**	3.27	1.00	0.00	0.00	26.1	26.1	15.0
		FFT	2.23	2.54	**1.14**	2.56	**1.15**	2.87	1.12	0.00	0.49	24.6	25.4	20.4
		IFFT	2.23	2.53	**1.14**	2.56	**1.15**	2.83	1.11	0.00	0.45	24.8	25.5	20.8
		GSM.dec	2.45	3.27	**1.34**	3.16	**1.29**	3.97	1.26	0.00	0.01	38.1	38.9	34.6
		GSM.enc	2.60	3.42	**1.31**	3.42	**1.31**	4.21	1.23	0.00	0.00	27.5	27.9	26.7
SPEC			**1.06**	**1.11**	**1.09**	**1.15**	**1.10**	**1.19**	**1.06**	**0.12**	**4.77**	**18.5**	**19.9**	**16.4**
MEDIABENCH			**2.07**	**2.52**	**1.23**	**2.43**	**1.19**	**2.77**	**1.15**	**0.01**	**1.44**	**27.9**	**29.4**	**27.0**
COMM			**2.05**	**2.38**	**1.16**	**2.28**	**1.11**	**2.53**	**1.11**	**0.00**	**1.04**	**31.4**	**32.9**	**28.0**
MI BENCH			**2.24**	**2.66**	**1.20**	**2.60**	**1.17**	**2.94**	**1.14**	**0.05**	**0.62**	**27.5**	**28.4**	**24.2**
ALL 78			**1.78**	**2.02**	**1.18**	**2.01**	**1.15**	**2.19**	**1.12**	**0.05**	**1.73**	**26.3**	**27.6**	**23.8**

Table 4.4: Microarchitectural Properties of Benchmarks in MiBench, averages of each suite, and all 78 programs.

processor sees a 17.6% speedup over a 3-wide processor, whereas a 4-wide processor sees a 15% speedup. For SPEC, mini-graph processing is just 1% shy of achieving the speedup of a 4-wide processor. However, for the remaining three suites, a 3-wide mini-graph processor has higher IPC than a 4-wide non-mini-graph processor.

Most benchmarks experience few instruction cache misses. Only 6 of 78 have a 0.1% miss rate, and only 15 of 78 benchmarks have an average 0.01% miss rate. This means that instruction cache amplification goes largely unnoticed at our default instruction cache size (32 KB). Because mini-graphs do not effect data memory access, benchmarks whose performance is limited by data cache effects (*e.g.*, *mcf*) are less affected by mini-graphs. The final three columns show mini-graph enabled reduction in several key pipeline operations—instructions fetched, issue queue entries allocated, and physical register writes—for a 3-wide processor. The columns show the percent reduction of book-keeping operations on a 3-wide mini-graph processor compared to a 3-wide non-mini-graph processor.

4.2 Performance Contribution Analysis

This section analyzes the contributions of the various benefits of mini-graph processing to performance improvement. Translating resource amplification to performance improvement is not a transparent process because amplifying resources affects performance only *indirectly*. *Direct* effects are those that lengthen or shrink the critical path through a program's dataflow graph. Mini-graph-induced serialization, for example, *directly* degrades performance by introducing new dependences that lengthen the dataflow graph. Amplifying resources *indirectly* improves execution time—insofar as resource limitations are actually a bottleneck to a program's execution. The fewer resources a processor has, the more resource contention hinders performance, and the more amplification yields performance improvement.

Mini-graph processing amplifies all pipeline stages (including execute if ALU

Pipelines are present) as well as the capacity of all structures that hold instructions and register values. For a 3-wide baseline processor, this amplification effect has been shown to improve performance, enabling a 3-wide mini-graph processor to outperform a 4-wide non-mini-graph processor on average—the former yielding a 17.6% gain over a 3-wide non-mini-graph baseline, the latter a 15% gain.

The experiments in this section show that bandwidth amplification contributes to performance more than does capacity amplification. More importantly, they show that mini-graph processing provides a performance improvement only when as many structures and stages possible are amplified. Amplifying only a *few* structures or stages quickly moves all performance bottlenecks to those resources which are unamplified; these new bottlenecks prevent the amplification benefits from improving performance. Only when all resources—both bandwidth and capacity—are amplified can the instructions flow through the pipeline with an increased throughput.

This section focuses on the effect of amplifying the following five resources:

- in-order bandwidth (fetch, decode, rename, commit)
- out-of-order bandwidth (schedule, execute, register read/write)
- instruction cache capacity
- reorder buffer and physical register file capacity (ROB/regfile)
- issue queue capacity

The first two resources are bandwidths. In-order pipeline stages are naturally amplified because they process mini-graph handles rather than constituents. The out-of-order schedule stage is amplified for the same reason; scheduling mini-graphs on ALU Pipelines and lookahead scheduling both allow scheduling at handle granularity. Execution is amplified by the additional execution resources of the ALU Pipeline. Physical register read/write is amplified because register values internal to a mini-graph are neither written to nor read from the register file.

The next three resources are capacities. Capacity amplification targets structures that hold instructions (*e.g.*, issue queue, ROB, instruction cache) as well as those that hold register values (*e.g.*, register file). Structures holding instructions are amplified because they hold handles rather than constituents. Structures holding register values are amplified because register values internal to a mini-graph do not consume entries.

Instruction cache occupancy is proportional to static instruction footprint. Amplifying instruction cache capacity reduces instruction cache misses. In this section, instruction cache capacity amplification is experimentally isolated from fetch bandwidth amplification, even though in practice these two are tightly-coupled with each other via the handle representation of the mini-graph in the instruction cache.

Reorder buffer and physical register file (*ROB/regfile*) capacity are proportional to in-flight instructions. By amplifying the ROB/regfile, a mini-graph processor can support more in-flight instructions. Issue queue occupancy is proportional to unexecuted in-flight instructions. By amplifying the issue queue, a mini-graph processor effectively has a larger number of instructions from which to schedule.

No single benefit is sufficient. The experiment in Figure 4.1 isolates the effect of each mini-graph benefit and illustrates the fact that *no single* mini-graph benefit is responsible for the performance improvement of mini-graph processing. Figure 4.1 begins by simulating a "no-benefit" mini-graph processor (grey line). A "no-benefit" mini-graph processor schedules and commits mini-graph constituents atomically, but the constituents are *never* represented by a handle in any physical structure. *None* of the five resources are amplified. Each mini-graph consumes one bandwidth and capacity slot for each constituent, rather than one slot for the handle. An n-instruction mini-graph consumes n entries in the instruction cache, ROB, issue queue, and register file. Any constituents scheduled for future cycles consume scheduling slots for those future cycles. Mini-graphs execute on singleton ALUs rather than ALU Pipelines. External and internal serialization constraints

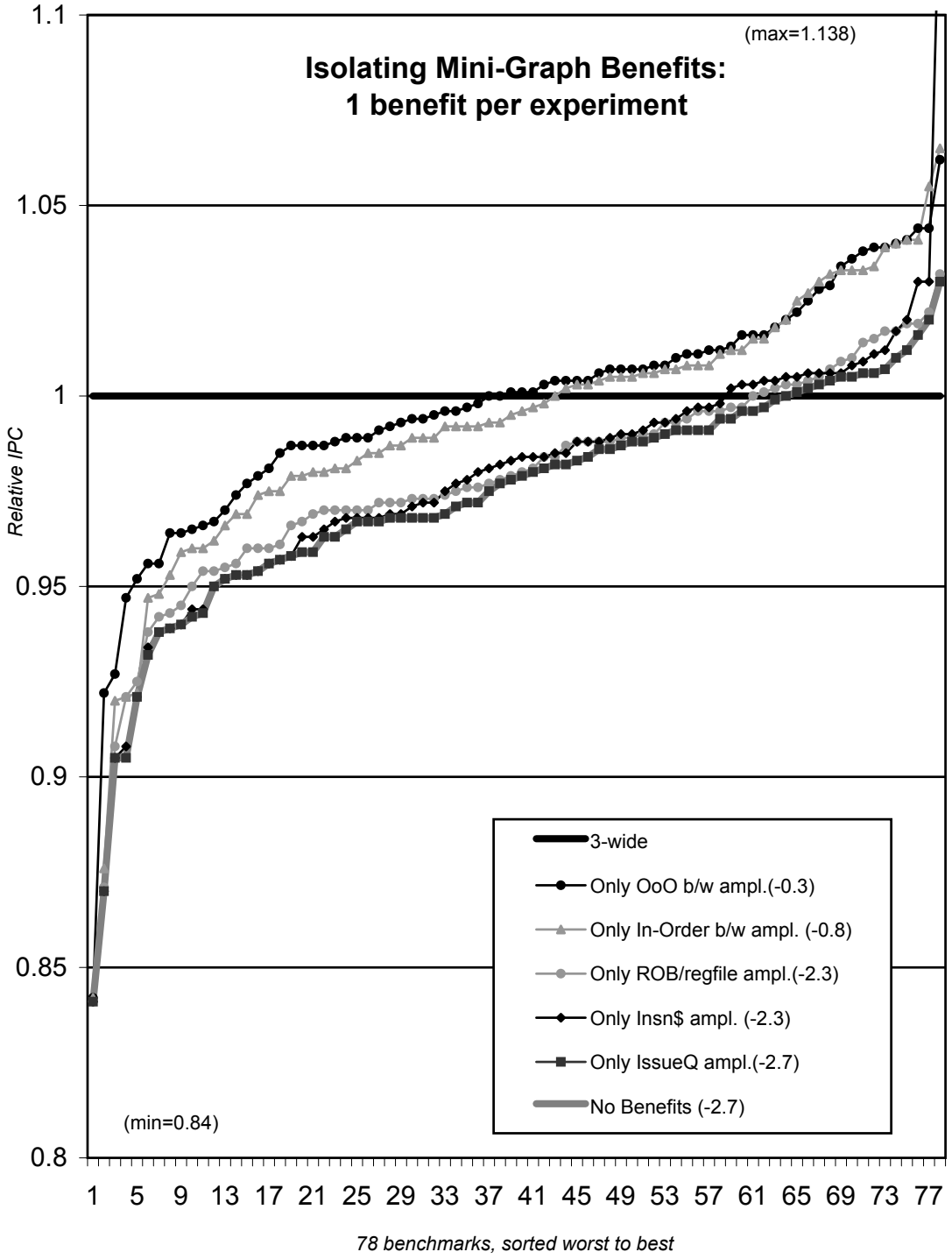


Figure 4.1: Lone Contributors of Mini-Graph Performance.

are still respected. When all mini-graph processing benefits are removed, performance improvement drops from drops from 17.6% to a 2.7% average performance loss over a 3-wide, non-mini-graph baseline processor (black line). This small average performance loss is analyzed briefly in the section following this one.

Figure 4.1 also simulates a series of mini-graph processors in which only a *single* benefit of mini-graph processing is enabled. In isolation, the most useful mini-graph benefit is out-of-order bandwidth amplification (black circle). Enabling out-of-order bandwidth amplification improves the "no-benefit" mini-graph processor, bringing its performance essentially back to the performance of the baseline non-mini-graph processor. Amplifying in-order pipeline stages (grey triangle) is the second most useful benefit, yielding a 2% performance improvement over the "no-benefit" mini-graph processor. Capacity amplification benefits make almost no performance impact when enabled in isolation.

When in- and out-of-order amplification benefits are combined, however, (*i.e.*, all pipeline stages are amplified) they yield an average 2.5% performance improvement over a baseline processor (not shown). In other words, although bandwidth amplification is the most useful benefit of mini-graph processing, supporting *only* bandwidth amplification does not approach the performance improvement seen by a fully-functional mini-graph processor. Conversely, when all forms of bandwidth amplification are *disabled*, all forms of capacity amplification combined yield an average 1.8% performance loss over a baseline processor.

Each mini-graph benefit in isolation pales in comparison to the full benefit) mini-graph processor which yields an average 17.6% performance improvement. In fact, no single benefit produces a performance improvement over the baseline non-mini-graph processor. This experiment shows that the performance gains of mini-graph processing cannot be attributed to the amplification of a single resource. Furthermore, although bandwidth amplification is more beneficial in isolation than capacity amplification, neither is sufficient by itself.

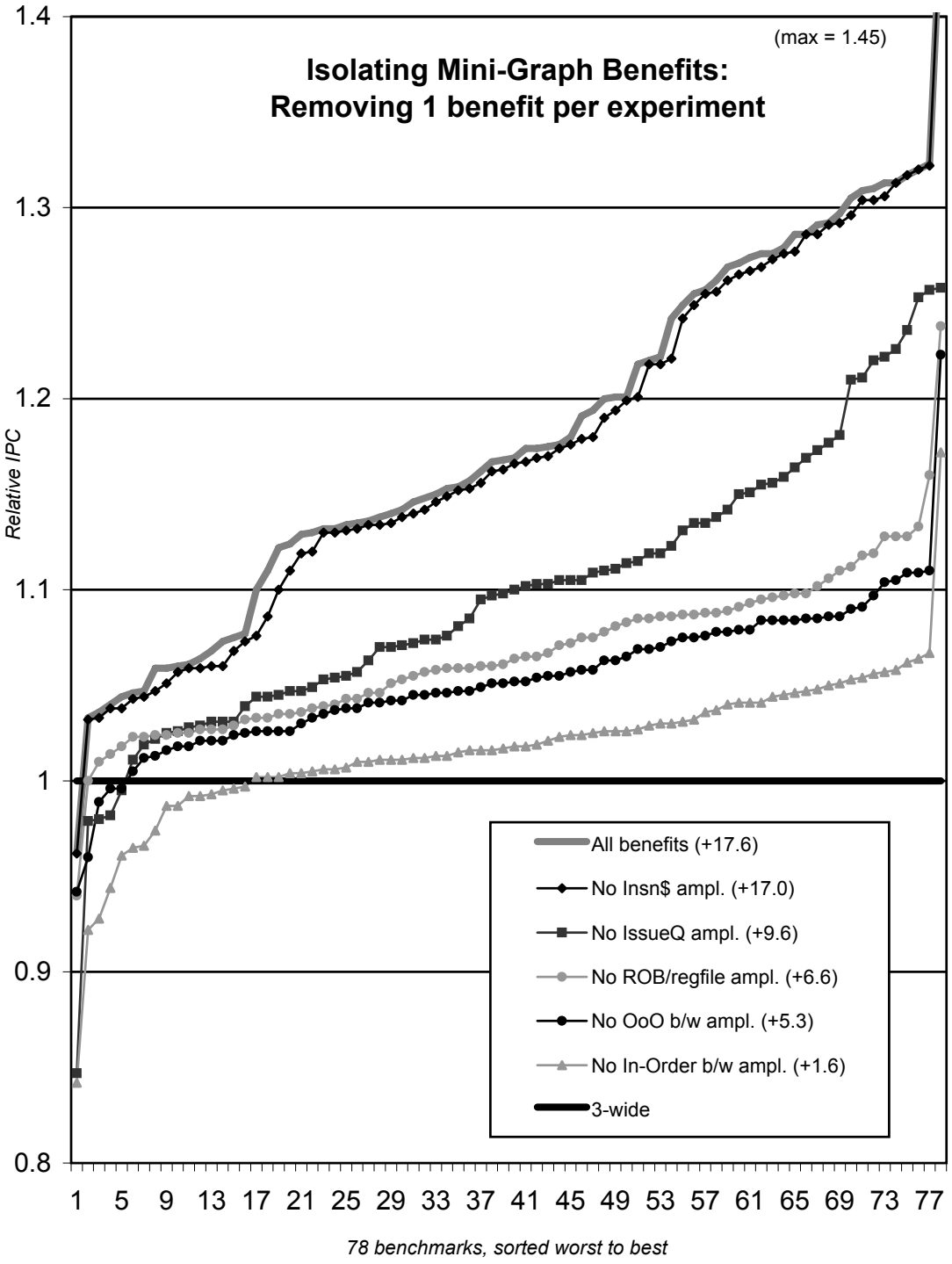


Figure 4.2: Isolating Components of Mini-Graph Performance Contributions.

Each benefit is necessary. The next set of experiments show the extent to which each individual resource amplification is necessary for the mini-graph processor performance. In Figure 4.2, each experiment shows a single mini-graph benefit *disabled*. The fourth experiment, for example, (*no OoO b/w ampl.*, black circle) shows the performance improvement of a mini-graph processor where all structures and stages are amplified *except* schedule and execute. In this case, over two-thirds of the performance improvement is lost because the scheduling and execution stages are not amplified. These stages become the new performance bottleneck.

The data in Figure 4.2 indicate that in-order bandwidth is the strongest performance limiter; *i.e.*, it is the resource that must be amplified before any other resource amplification can translate to performance improvement. This is unsurprising. If the in-order front-end cannot supply the out-of-order core with an increased number of instructions, the amplified out-of-order core cannot be utilized. Out-of-order bandwidth is the next most important contributor to performance. Capacity is once again a secondary contributor.

Instruction cache capacity amplification has a minimal effect on average. This is also unsurprising, considering that most of the benchmarks have low instruction cache miss rates in the first place (see Tables 4.3 and 4.4 of Section 4.1). One notable exception to this trend is the MI Bench program *network.patricia*, which normally has a 1.32% instruction cache miss rate. By amplifying only the instruction cache, the instruction cache miss rate decreases to 0.74%, yielding a performance improvement of 13.8%. Section 4.4 revisits the benefits of instruction cache capacity amplification by considering mini-graph processors with smaller instruction caches.

Analysis of "no-benefit" mini-graph processing. This section analyzes the 2.7% performance loss of the "no-benefit" processor shown in Figure 4.1. Once again, a "no-benefit" mini-graph processor schedules and commits mini-graph constituents atomically, but the constituents themselves are not represented by handles

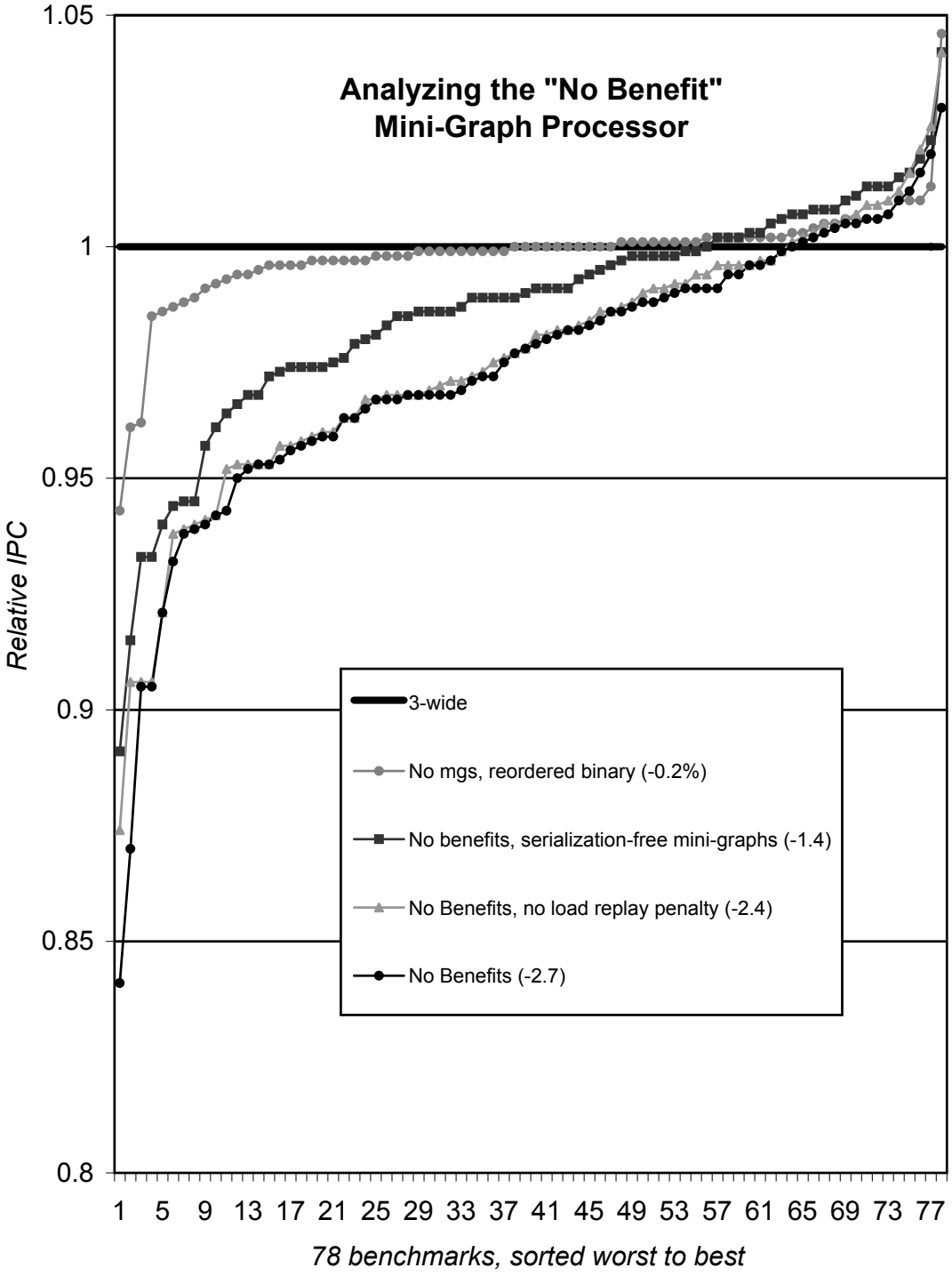


Figure 4.3: Analyzing the "No Benefit" Mini-Graph Processor.

and therefore offer no capacity or bandwidth amplification. Scheduling and committing constituents atomically has several negative performance consequences that explain the observed performance loss. Before addressing the performance implications of atomic scheduling and commit, however, this section first rules out several secondary performance effects—other plausible but in practice *not* culpable performance limiters associated with the "no-benefit" mini-graph processor.

Figure 4.3 shows a series of experiments, all relative to a 3-wide non-mini-graph baseline (black line). The first experiment shows the "no-benefit" mini-graph processor from Figure 4.1, in which all amplification benefits are disabled (black circle). The remaining three s-curves, in grey, rule out three possible causes of the performance loss.

The first possible performance effect is that of reordering the static instructions to make mini-graph constituents contiguous. Static reordering changes both instruction cache placement and dynamic instruction order. Creating or removing a potential instruction cache conflict within a frequently executed section of code can lead to performance loss or gain. Implicitly prioritizing or de-prioritizing critical instructions can similarly alter the performance of a program. The *reordered binary* s-curve of Figure 4.3 (grey circle) shows the performance effect of reordering each program binary to make mini-graph constituents contiguous but then running the binary without mini-graphs on the non-mini-graph baseline processor. Although individual benchmarks show performance variation from 5.7% loss to 4.6% gain, the effect of instruction reordering on average is minimal, yielding a 0.2% performance loss over a baseline processor.

The second possible performance effect is the requirement that when a non-terminal load in a mini-graph misses in the data cache, the entire mini-graph must be replayed. Because mini-graphs are atomic, there is no way to replay only some of the mini-graph constituents. The *no load replay penalty* s-curve of Figure 4.3 (grey triangle) artificially removes this requirement in order to see the extent to which this

penalty contributes to the performance loss of the no-benefit mini-graph processor. Once again, a few benchmarks improve markedly (*mcf*, for example, goes from a 16% performance loss to a 9% performance loss), but on average, performance improves only slightly from a 2.7% loss to a 2.4% loss.

A third possible performance effect is serialization. Serialization *does* stem from the restriction that mini-graph constituents be scheduled atomically, but the mini-graphs shown in these experiments are selected with the serialization-aware selection algorithm ($\text{Slack}_{\text{Profile}}$). $\text{Slack}_{\text{Profile}}$ heuristically removes harmful—but not all—serializing mini-graphs, and should therefore remove the serialization penalty associated with atomic scheduling. To support the claim that serialization is not the cause of the observed performance loss, Figure 4.3 shows a *serialization-free mini-graph* s-curve (grey square) that uses mini-graphs selected with $\text{Struct}_{\text{None}}$. $\text{Struct}_{\text{None}}$ is a selection algorithm that forbids all forms of serialization in mini-graphs. The processor remains at its no-benefit configuration. The result is a 1% performance improvement over the original no-benefit processor using mini-graphs selected by $\text{Slack}_{\text{Profile}}$. This s-curve still shows an average 1.4% performance loss over a baseline processor; serialization cannot be the sole reason for the performance loss. Furthermore, because both performance loss *and mini-graph coverage* drop by approximately 40%, in all likelihood, performance improves not because harmful serialization is removed, but because the number of mini-graphs in play decreases when using the more restrictive selection algorithm.

As a side note, the average 2.7% performance loss substantiates the claim that serialization-aware mini-graph selection is effective: even when mini-graph processing offers *no* amplification benefits whatsoever, performance loss is minimal.

Once instruction ordering, load replay penalties, and serialization are ruled out as possible causes of the performance degradation, the two actual causes remain. First, scheduling and committing multiple instructions atomically increases structure occupancy and reduces effective capacity. Second, reserving future scheduling

slots without increasing scheduling bandwidth yields a scheduling priority inversion. In both cases, the "no-benefit" mini-graph processor incurs the overhead of fusion without any of the benefits of fusion.

Increased structure occupancy and pipeline hiccups. The capacity of structures that hold instructions is effectively reduced because mini-graph instructions are still allocated and freed *en masse*. A 4-instruction mini-graph, for instance, not only requires the allocation of four ROB entries (because ROB amplification is disabled) but also holds onto these four entries for a longer period of time. Only after all four instructions are completed can all four ROB entries be freed. This creates a "bursty" occupancy of structures that can translate to increased congestion which degrades performance.

Atomic processing also creates pipeline stalls. The most significant example of this occurs between the rename and issue stages. Typically, instructions pass through each pipeline stage atomically; they are renamed in one cycle and ready for scheduling in the following cycle (modulo the rename latency). A 4-operation mini-graph traveling down a 3-wide pipeline with no amplification benefits, however, cannot be renamed in its entirety in a single cycle. The fourth instruction will be renamed in the cycle after the first three instructions. Meanwhile, mini-graph constituents wait until all constituents are renamed before scheduling can take place. The same effect occurs for smaller mini-graphs that simply happen to "straddle" the pipeline width boundaries. This additional pipeline stall is the primary cause of performance degradation.

Figure 4.4 shows an example of how both structure capacity and pipeline stalls are affected by no-benefit mini-graph processing. Figure 4.4a shows a sequence of six instructions, `A-F`. Instructions `B-E` are dependent on one another in a chain as shown by the abstract dataflow graph. In later parts of this figure, instructions `B-D` are fused into a mini-graph; in Figure 4.4, however, they remain singletons. Instructions

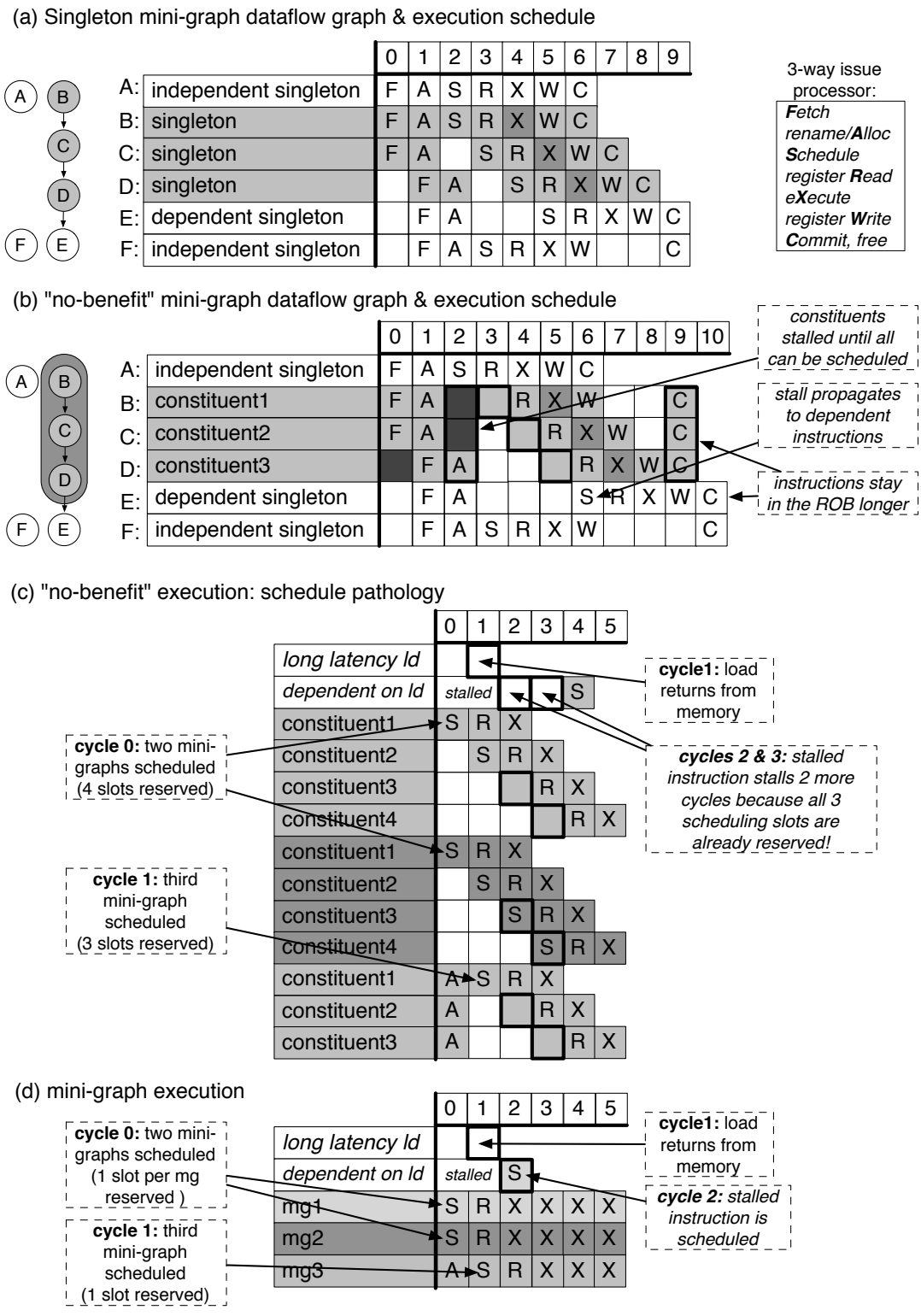


Figure 4.4: "No Benefit" Mini-graph Overhead

A and F are completely independent of all instructions shown. The processor has a 3-wide pipeline; instructions A-C are fetched first, followed by D-F. The entire sequence finishes execution at cycle 7 and all instructions commit by cycle 9.

Figure 4.4b shows the same six instructions, but with instructions B-Dplaced in a mini-graph. Because this is a no-benefit mini-graph processor, each constituent still consumes bandwidth and structure slots. Furthermore, the mini-graph constituents must be scheduled as a unit. At cycle 2, the mini-graph cannot be scheduled because instruction D is not yet renamed. This stall delays all dependent instructions. At cycle 3, D has been renamed, and the three constituents are scheduled. The sequence previously completed execution at cycle 7; it now completes execution at cycle 8. The structures also experience increased and bursty occupancy. Instructions B-F previously left the ROB over a period of 4 cycles, committing from cycle 6 to 9. These same instructions now remain in the ROB for longer and leave the ROB in a large burst at cycles 9 and 10.

Scheduling inversions. The second reason why the performance of the no-benefit mini-graph processor is sub-baseline is due to a scheduling priority inversion that occurs in benchmarks with high mini-graph coverage. As shown in Figure 4.4b, when a mini-graph is scheduled (at cycle 3), future scheduling slots are consumed for each constituent. When many mini-graphs are scheduled, these future reservations can actually use up *all* available scheduling slots for a particular future cycle. Older instructions that finally have inputs ready are not given the age priority they normally would because younger instructions "beat them" to the scheduling resources. Unfortunately, it is these old instructions waiting for inputs that tend to be the critical instructions of a program. Delaying them–even by just one or two cycles—very quickly degrades performance.

Figure 4.4c shows an example of this scheduling pathology. The first instruction is a long-latency load to memory. The second instruction depends on this load and is waiting to be scheduled. In cycle 1, a 3-instruction mini-graph is scheduled,

consuming three scheduling slots. Between these three mini-graphs all available scheduling slots for cycles 2 and 3 have been consumed. When the load returns from memory at cycle 1, the instruction dependent on it must wait until cycle 4 to be scheduled. Note that this pathology only happens on the no-benefit mini-graph processor. In an actual mini-graph processor where each mini-graph consumes only one scheduling slot, the dependent instruction faces no such scheduling delay (see Figure 4.4d).

Both of the above problems—scheduling delays and priority inversions—are felt more acutely the more mini-graphs there are. Unsurprisingly, the with the worst no-benefit performance are those with the highest mini-graph coverage. The MI Bench program *consumer.tiff*, for example, is the worst-performing benchmark (-13%) on the no-benefit processor and is also one of only two benchmarks with coverage higher than 50%. Both of these problems are specific to the "no-benefit" mini-graph processor. They prevent a no-benefit mini-graph processor from yielding baseline performance, but they in no way affect the performance of a mini-graph processor in which amplification does take place.

Summary: Purpose of these analyses. It is important to note that analyzing the performance loss associated with a no-benefit mini-graph processor is motivated by the desire to understand the results and any insights they may offer into possibly unknown effects of mini-graph processing. The "no-benefit" mini-graph processor exists only as an investigational tool; this dissertation proposes the mini-graph processor, not the no-benefit version thereof.

Similarly, teasing out the amplification benefits of mini-graph processing is an investigational endeavor geared at understanding how different forms of amplification translate and contribute to performance improvement. It is not motivated by trying to decide which stages or structures a mini-graph processor *should* amplify. A mini-graph processor that does not amplify the ROB, issue queue, and most pipeline stages, would have to operate on constituents rather than handles. A mini-graph

processor that does not amplify the physical register file would have to rename at a constituent granularity in order to allocate more than one output register. Any of these modifications would require fundamental changes to the hardware. The only forms of amplification for which this is not the case are instruction cache or execution stage amplification. It would be possible to use a different mini-graph encoding scheme which does not create instruction cache capacity amplification. Similarly, a mini-graph processor could be created without ALU Pipelines.

4.3 In-Order Performance Analysis

This section analyzes mini-graph performance improvement on an in-order machine. In a dynamically scheduled processor, both capacity and bandwidth amplification play an important role in performance improvement. An in-order processor, however, has different resource limitations and performance characteristics. Schedule and execute—the out-of-order stages on a dynamically scheduled processor—are in-order. Some out-of-order structures—*e.g.*, the issue queue, ROB, and (rename) register file—do not exist in an in-order processor.

Figure 4.5 shows mini-graph performance for an out-of-order processor (left) and an in-order processor (right). The in-order processor matches the now-familiar out-of-order processor in all common components (3-wide, same memory hierarchy, same branch predictor, *etc.*). The out-of-order machine has an 80% performance advantage over its in-order counterpart (not shown). Each graph uses the respective non-mini-graph processor for its baseline. The out-of-order mini-graph processor shows an average 17.6% performance improvement. The in-order mini-graph processor shows an average 15.0% performance improvement.

Figures 4.6 and 4.7 repeat the experiments of Figures 4.1 and 4.2—in which only one mini-graph processing benefit is active, followed by each benefit being removed one at a time—for an in-order processor. Issue queue and ROB/register amplification

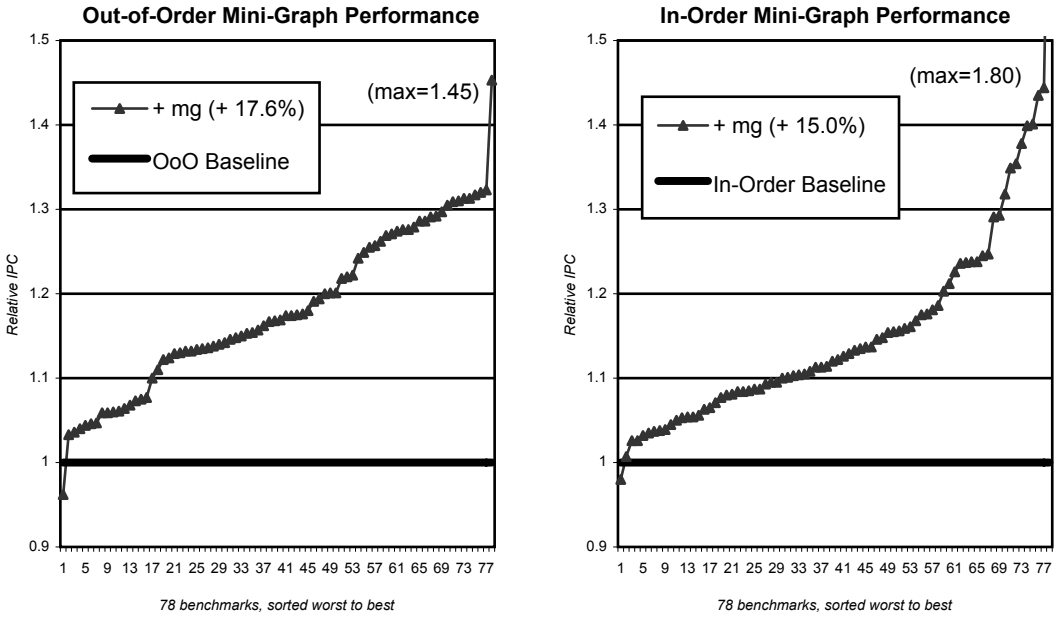


Figure 4.5: Mini-graph performance in out-of-order (left) and in order (right) contexts.

are no longer shown because their capacities play no performance role in an in-order machine. Instruction cache capacity amplification remains. Whereas the previous section makes an in-order versus out-of-order distinction between the various pipeline stages, this section simply distinguishes between the front-end pipeline stages (fetch and decode) and the issue and execute pipeline stages.

As in the previous section, instruction cache capacity amplification continues to have limited utility at the present processor configuration. In the in-order processor, fetch and decode bandwidth amplification is the strongest effect. Amplifying these front-end stages in isolation (shown in Figure 4.6) yields a 12.5% performance improvement over an in-order, non-mini-graph processor. The same is *not* true of the out-of-order processor in which amplifying the scheduling and execution bandwidth is most beneficial in isolation. In an in-order processor, the issue and execution stages are more tightly coupled with the front-end. Instructions do not pool in an issue queue awaiting dynamic scheduling as they do in an out-of-order processor.

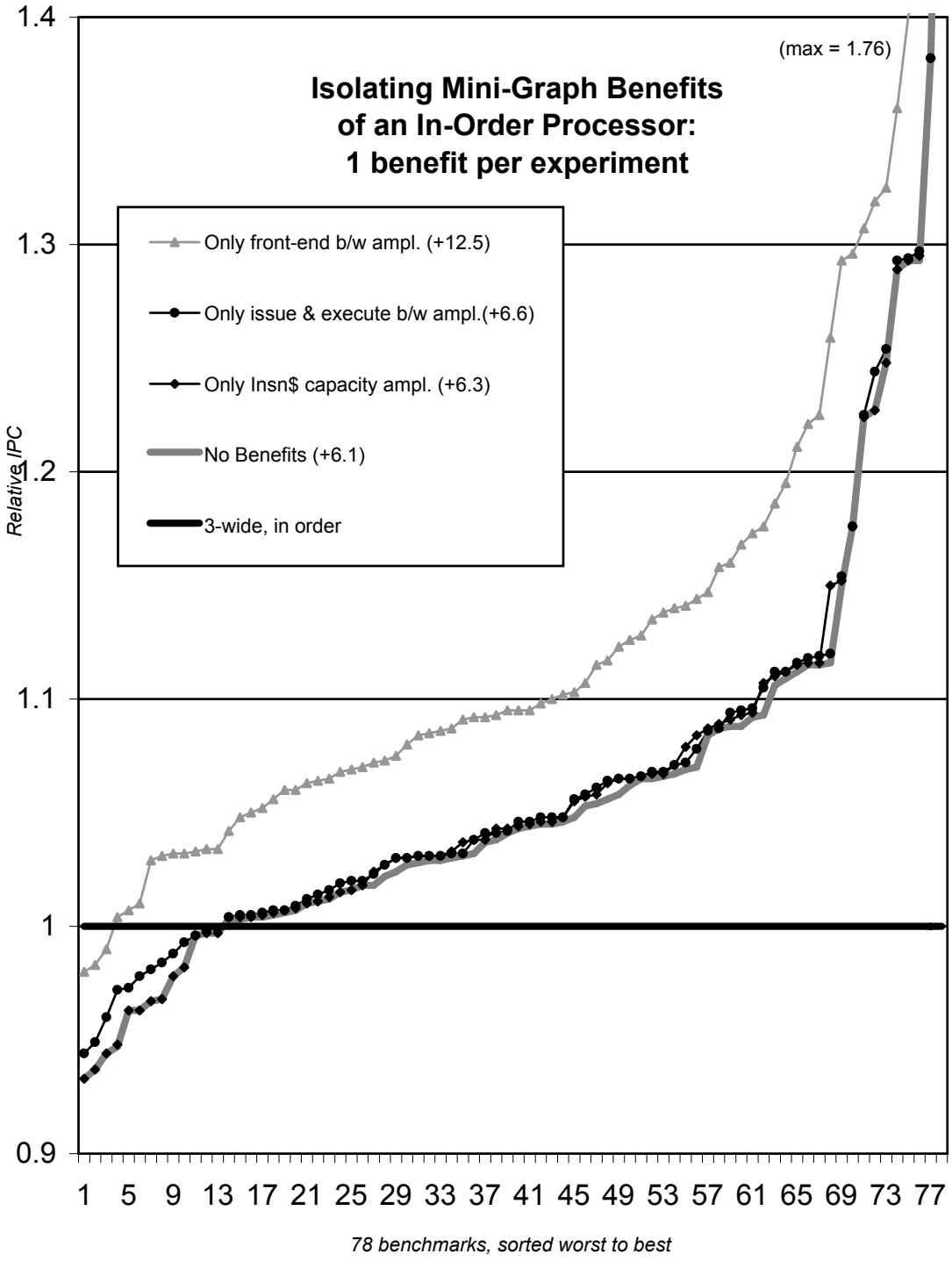


Figure 4.6: Lone Contributors of In-Order Mini-Graph Performance

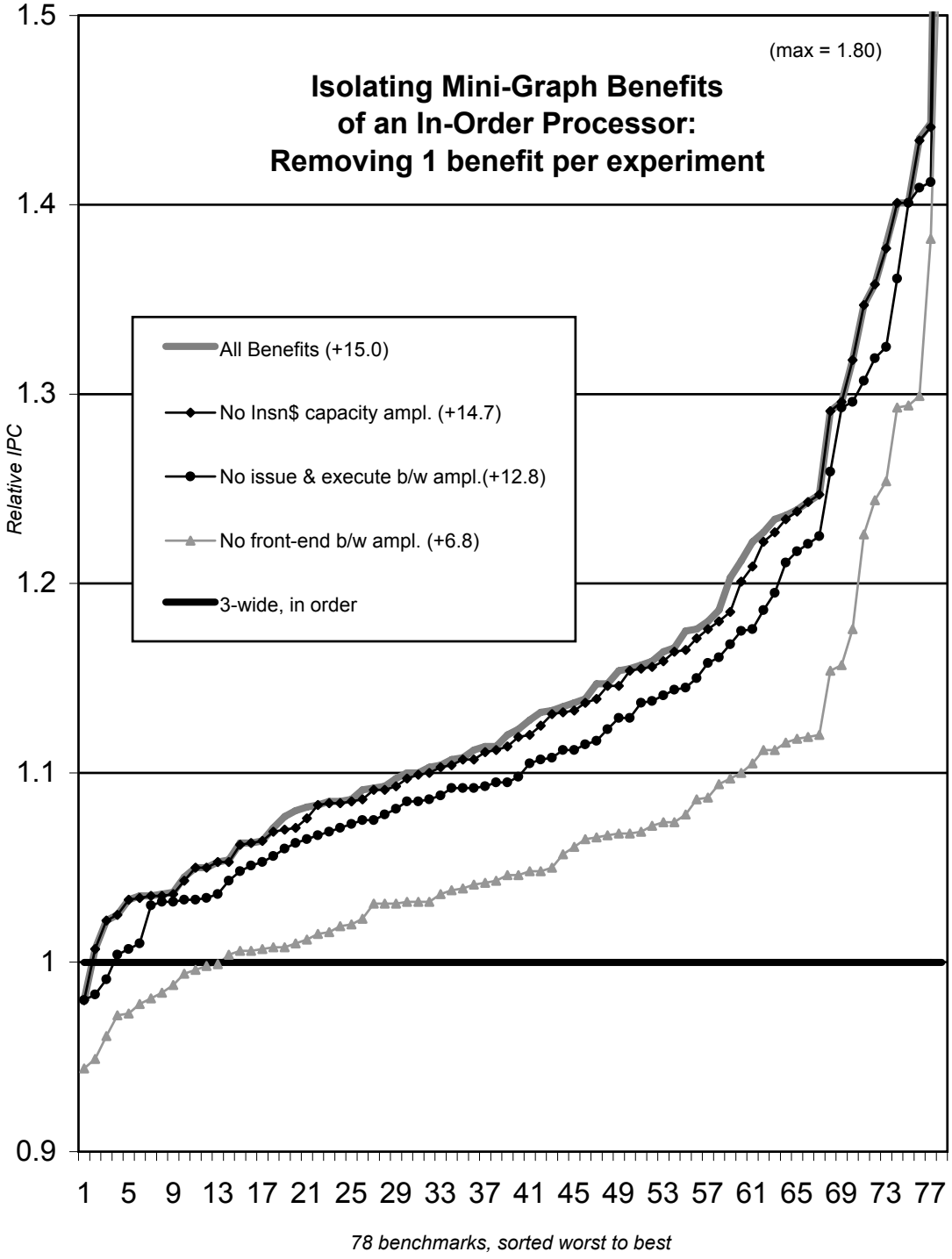


Figure 4.7: Isolating Components of In-Order Mini-Graph Performance

Not amplifying fetch and decode (shown in Figure 4.6) has the most debilitating effect on the in-order mini-graph processor. This phenomenon is also seen on the out-of-order processor (shown in Figure 4.2). Not amplifying the front-end stages of any processor limits throughput by effectively pinching beginning of an otherwise amplified pipeline.

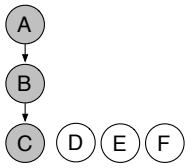
A major difference between the in-order and out-of-order experiments shown here is that the "all-or-nothing" amplification-to-performance behavior no longer applies. In the out-of-order processor, amplifying as many structures and stages as possible is critical to achieving the 17.6% performance improvement. In an in-order processor, simply amplifying front-end bandwidth achieves a 12.5% speedup—almost 80% of the performance improvement offered by a full benefit) in-order mini-graph processor. In other words, a single benefit—front-end bandwidth amplification—is sufficient by itself to achieve good performance improvement. This is decidedly *not* the case for the out-of-order processor.

Forgoing all mini-graph benefits. The most surprising result in Figure 4.6 is the performance of the "no-benefit" in-order mini-graph processor. For the out-of-order machine, removing all amplification benefits yields a 2.7% performance loss over a baseline processor. For an in-order machine, "no-benefit" mini-graph processing still achieves a 6.1% performance gain. What this exposes is a subtle and intrinsic benefit of mini-graph execution that proves especially helpful to an in-order processor. The benefit can best be described as local, out-of-order execution.

The in-order processor's main bottleneck is its strict adherence to program order at the issue stage. A mini-graph, however, is treated as a single instruction with respect to issue. Chains of instructions statically fused into mini-graphs are issued as a unit and executed back-to-back in the following cycles. Surrounding instructions are issued in order with respect to the handle, but they have the potential to *execute* out of order with respect to the mini-graph constituents.

Figure 4.8 shows an example. Figure 4.8a shows six instructions executed as

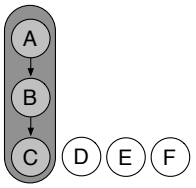
(a) Singleton dataflow graph & execution schedule on an in-order machine



		0	1	2	3	4	5	6	7	8	9
A:	singleton	F	D	S	R	X	W	C			
B:	singleton	F	D		S	R	X	W	C		
C:	singleton	F	D			S	R	X	W	C	
D:	independent singleton		F	D		S	R	X	W	C	
E:	independent singleton		F	D		S	R	X	W	C	
F:	independent singleton		F	D			S	R	X	W	C

3-way issue processor:
***F**etch*
***D**ecode*
***iS**sue*
*register **R**ead*
*e**X**ecute*
*register **W**rite*
*Retire (**C**)*

(b) "No-benefit" mini-graph dataflow graph & execution schedule on an in-order machine



		0	1	2	3	4	5	6	7	8	9
A:	singleton	F	D	S	R	X	W	C			
B:	singleton	F	D		S	R	**X**	W	C		
C:	singleton	F	D			S	R	**X**	W	C	
D:	independent singleton		F	D	S	R	**X**	W		C	
E:	independent singleton		F	D	S	R	**X**	W		C	
F:	independent singleton		F	D		S	R	**X**	W		C

(c) Mini-graph execution schedule on an in-order machine

		0	1	2	3	4	5	6	7	8	9
ABC:	mini-graph	F	D	S	R	**X**	**X**	**X**	W	C	
D:	independent singleton	F	D	S	R	**X**	W			C	
E:	independent singleton	F	D	S	R	**X**	W			C	
F:	independent singleton		F	D	S	R	**X**	W			C

Figure 4.8: Local Out-of-Order Execution Effect of Mini-graphs

singletons on a 3-wide, in-order processor. Instructions `A`, `B`, and `C` are dependent on one another and can be placed in a mini-graph. Instructions `D`, `E`, and `F` are independent of all other instructions shown. All instructions execute in a single cycle. The instructions are fetched in two batches of three instructions each. The dependences between instructions `A`, `B`, and `C` introduce issue stalls. The entire sequence takes 9 cycles to complete.

Figure 4.8b shows these same six instructions executed on a "no-benefit" in-order mini-graph processor. Instructions `A`, `B`, and `C` are formed into a mini-graph, but the only benefit they offer is the local, out-of-order execution phenomenon being described here. At cycle 2, instructions `A`, `B`, and `C` are issued, consuming issue slots for cycles 2, 3, and 4; there is no issue amplification on the no-benefit processor. Issuing all three constituents at once is a task only possible on the mini-graph processor; the non-mini-graph processor can only issue each instruction as its inputs become available. As a result, as early as cycle 3, the next three instructions can also be issued. The local, out-of-order execution effect is outlined in cycles 5 and 6. Although they are issued after instruction `C`, instructions `D` and `E` actually execute *before* instruction `C`. Although the entire sequence *completes* in the same number of cycles as in Figure 4.8a, instructions `D`, `E`, and `F` all finish execution 1 cycle earlier.

On a (full-benefit) mini-graph processor, the benefit is even greater because all pipeline stages are amplified by virtue of operating on handles and executing on ALU Pipelines. This is shown in Figure 4.8c. Here, instructions `D`, `E`, and `F` all finish 2 cycles earlier than they do in the singleton execution of Figure 4.8a. The local, out-of-order execution effect of mini-graphs is the same one produces by vectorization: multiple operations are fused into a single instruction which take up fewer pipeline resources and around which execution reordering can take place. An in-order mini-graph processor benefits particularly from this effect because it is normally so constrained by instruction order.

It could be argued that the instruction sequence seen in Figure 4.8a—constructed

as an easy-to-follow example—would never be produced by a compiler targeting an in-order processor. A compiler that targets an in-order processor will interleave the dependent instructions with the independent ones. While it is true that the baseline in-order performance might be aided by a different compiler, it is *not* true that a compiler can match the local, out-of-order execution effect of mini-graphs. Static instruction reordering cannot achieve the same results as dynamic instruction reordering. For example, if instructions `A-C` are in a different basic block than instructions `D-F`, a compiler would produce exactly that instruction sequence because interleaving would require basic block merging or predication or the like. When there are not enough independent instructions, a dependence chain will still produce the stalls shown in Figure 4.8a.

As a side note, the local, out-of-order execution effect of an in-order mini-graph processor implicitly amplifies the execution stages; this is an effect that cannot be "turned off." This helps explain why the "all-or-nothing" performance of amplification (measured in Figure 4.6) seems not to apply to the in-order processor. The s-curve that only amplifies the front-end (grey triangle) in fact does amplify all pipeline stages to some extent due to the presence of the local, out-of-order execution effect. Similarly, the seeming ineffectiveness of amplifying only the issue and execution stages (black circle) is because the "no-benefit" processor already has a substantial amplification of both issue and execution in the first place.

The performance loss of a "no-benefit" out-of-order mini-graph processor (Section 4.2) is due to the combination of not amplifying certain pipeline stages but still requiring mini-graphs to proceed through those stages atomically. These artifacts disappear on a mini-graph processor that amplifies the usual stages and structures. In contrast, the performance benefit of the "no-benefit" in-order mini-graph processor is real. Local out-of-order execution would exist on an in-order (full-benefit) mini-graph processor and contributes significantly to its performance gains.

4.4 Exploiting Instruction Cache Amplification

All experiments so far use a 32 KB, 32-byte line-size, 4-way associative instruction cache. As shown in Tables 4.3 and 4.4, this configuration yields an instruction cache miss rate of less than 0.01%. As a consequence, the performance contribution of instruction cache capacity amplification in mini-graph processing is small. As shown in Figure 4.2, removing instruction cache capacity amplification decreases the performance improvement from 17.6% only to 17.0%.

In this section, we explore the potential of using mini-graph processing to compensate for reduced instruction cache capacity. If mini-graphs enable reduced instruction cache capacity in a robust way, then the instruction cache can be physically reduced to save area. If the enabled reduction is less robust, then techniques like selective cache ways [3] can be used to opportunistically reduce power.

In order to explore instruction cache amplification, this section separates the SPEC suite from the other three suites (MediaBench, Comm Bench, and MI Bench, hereafter referred to as the *MCM suite*) and maintains two separate baseline instruction cache sizes for each suite. This separation is motivated by the instruction cache miss rates shown in Figure 4.4. The SPEC suite requires a larger instruction cache than the MCM suite. An instruction cache sized for SPEC will be so over-provisioned for MCM that amplification will have no effect. An instruction cache sized for MCM will create such instruction cache pressure for SPEC that the amplification benefits of mini-graph processing will be unfairly inflated.

Figure 4.9 shows the result of shrinking the instruction cache size from 32 KB to 8KB by reducing the number of sets in the cache while holding line size and associativity constant. The data motivate using 32 KB as the default instruction cache size for SPEC and 24 KB as the default size for MCM.

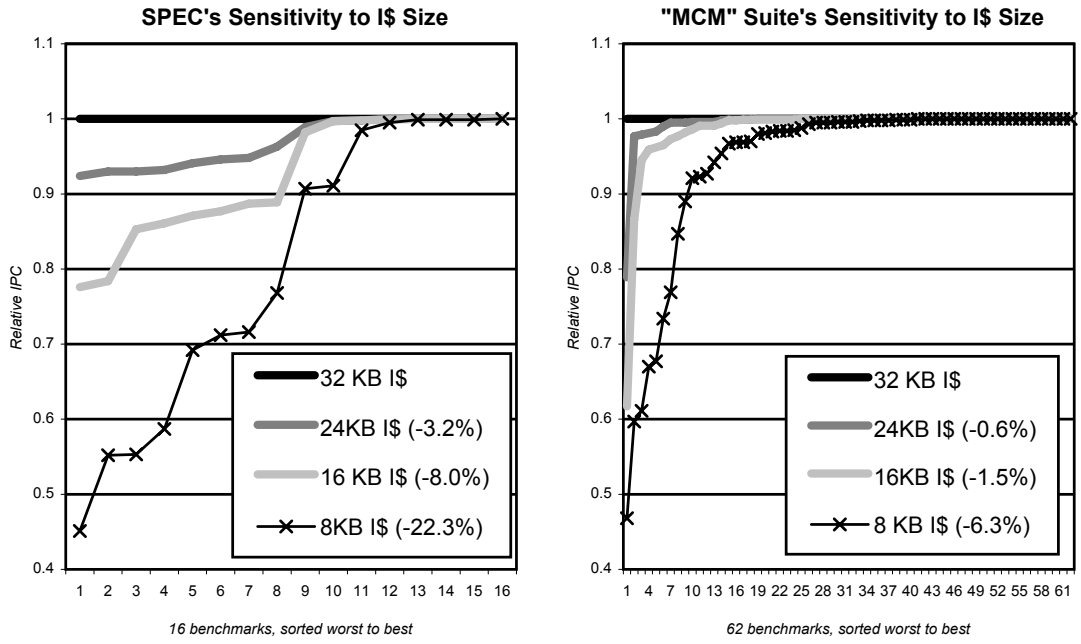


Figure 4.9: Instruction Cache Behavior Differences. Performance degradation associated with shrinking the instruction cache from 32 KB to 8 KB in 8 KB increments. Left: SPEC benchmarks. Right: "MCM" benchmarks (MediaBench, CommBench, and MI Bench)

4.4.1 Instruction Cache Amplification for SPEC

Figure 4.10 begins with a 3-wide non-mini-graph baseline (the $y = 1$ axis) and (for reference) a 4-wide non-mini-graph performance curve that shows a 10.1% performance improvement for the SPEC benchmark suite. Three mini-graph experiments (*diamonds*) show the 3-wide mini-graph performance across three different instruction cache sizes: 32 KB (*dark*), 24 KB (*medium*), and 16 KB (*light*). The 32 KB instruction cache (*dark diamond*, 8.8%) is the configuration used in all mini-graph experiments previous to this section and plots the data shown in Table 4.3 for SPEC.

The 3-wide mini-graph processor with the 32 KB instruction cache yields a 8.8% performance improvement—just 1.3% shy of the 10.1% speedup of the 4-wide processor. By shrinking the instruction cache by 8 KB (from 32 to 24 KB) a 6.2% speedup is still achieved. Furthermore, only one of the sixteen benchmarks experiences a new performance slowdown over the 3-wide 32 KB instruction cache baseline. (This does not count *mcf*, which experiences a slowdown on *all* mini-graph configurations because $\text{Slack}_{\text{Profile}}$ does not incorporate load miss predictions.) Compare this to the SPEC graph on the left of Figure 4.9 which shows that decreasing the instruction cache size to 24 KB yields slowdowns in half the benchmarks. However, once the instruction cache is reduced to 16 KB (*light diamond*) almost half the benchmarks experience slowdowns over the baseline configuration.

Using some of the hardware budget of the instruction cache for the MGT introduces a loss of 2.6% to 5.8%, depending on the reduction to the instruction cache. For SPEC, instruction cache capacity amplification is not robust enough to enable physical reduction and area savings.

4.4.2 Instruction Cache Amplification for MCM

Figure 4.11 repeats the experiments shown in Figure 4.10, but for the MCM suite, with a 24 KB instruction cache baseline. The performance trend seen here is like that of Figure 4.10: each mini-graph processor with a smaller instruction cache

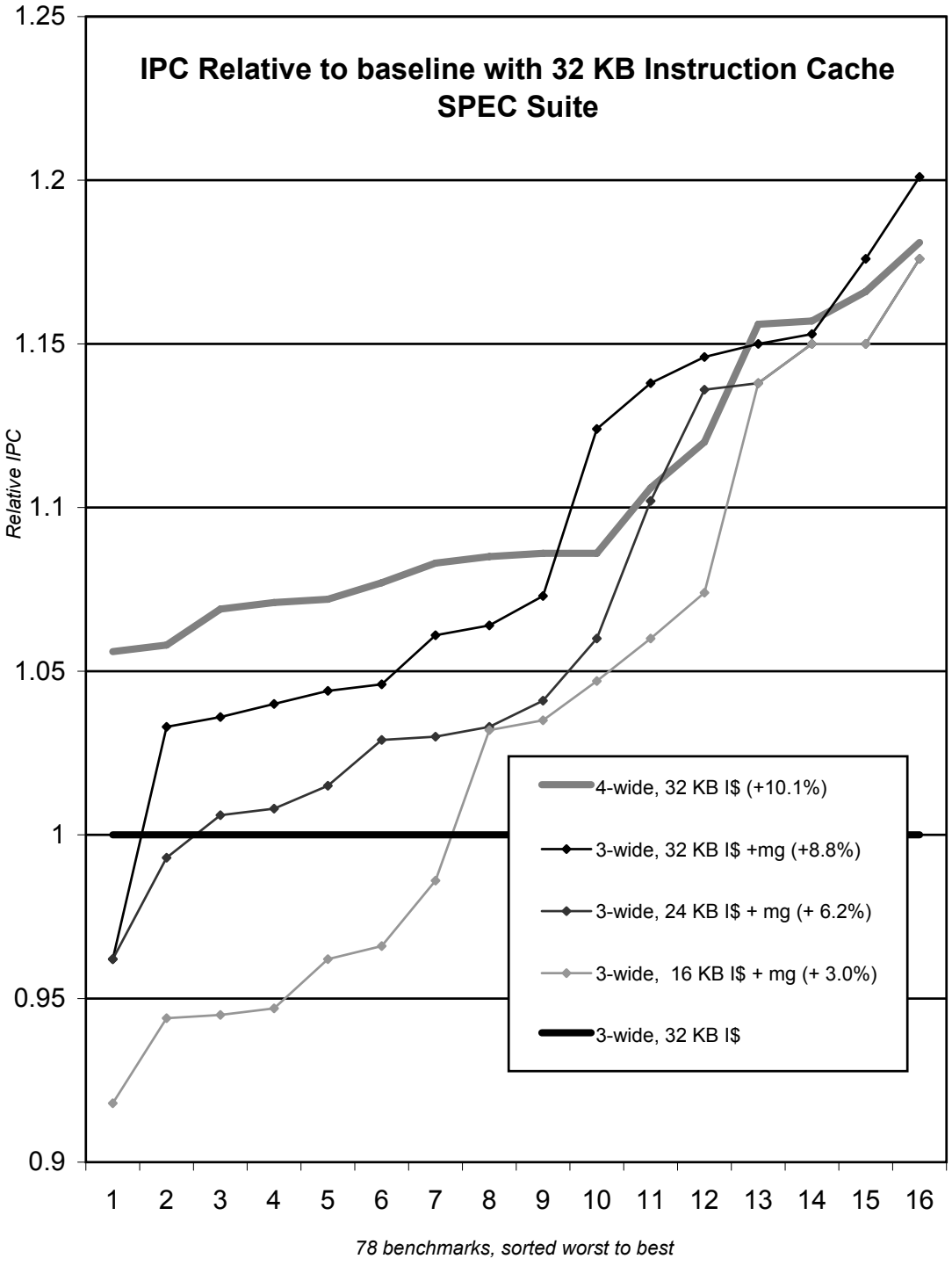


Figure 4.10: Mini-Graph Performance with Various Instruction Cache Sizes, SPEC. IPC relative to a 3-wide baseline processors with 32 KB instruction cache.

performs slightly worse, achieving performance improvements of 19.9%, 19.3%, and 14.8% for instruction cache sizes of 24, 16, and 8 KB, respectively, over the 3-wide non-mini-graph baseline.

Compared to the SPEC results, the MCM performance cost of a reduced instruction cache is smaller. Whereas the performance benefits of mini-graph processing for SPEC decrease by over 50% when the instruction cache is reduced from 32 to 16 KB, the benefits for MCM are reduced by only 25% when the instruction cache is reduced from 24 to 8 KB. Even more importantly, only a few MCM benchmarks experience a slowdown over the baseline processor when the instruction cache capacity is reduced. It should also be noted that two of the three mini-graph processor configurations out-perform the 4-wide non-mini-graph processor. This is because the MCM suite has more parallelism and higher coverage and benefits more from mini-graph processing in the first place.

For MCM, instruction cache capacity amplification effects do enable physical reduction and area savings. In the reduced instruction cache configurations of Figure 4.11, the mini-graph processors would have a 16 KB and 8 KB SRAM budget for the MGT. As will be seen in the next section, performance improvements comparable to and even better than a 4-wide processor can be achieved starting at MGT sizes of just 2 KB.

4.5 MGT Configuration and Area Analysis

This section explores the coverage and performance implications of various MGT configurations. Specifically, the coverage and performance "knee" of possible MGT configurations argues for the support of 3-stage ALU Pipelines and an area budget of between 1 KB and 6 KB, depending on the processor's benchmark targets and performance goals. Section 2.1 briefly discusses the implementation and coverage impact of reducing the number of MGT banks from seven to five. This section

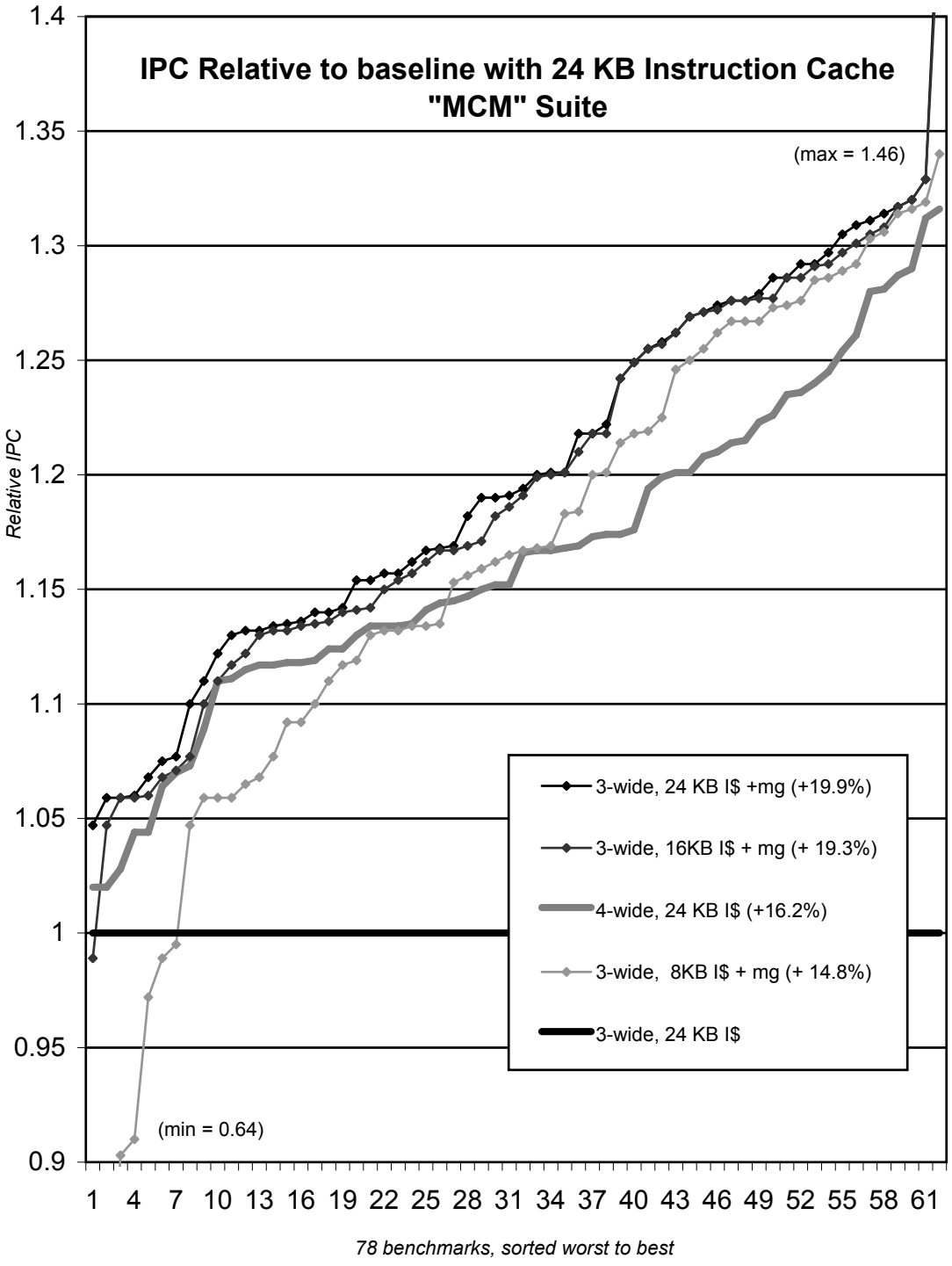


Figure 4.11: Mini-Graph Performance with Various Instruction Cache Sizes, MCM. IPC relative to a 3-wide baseline processors with 24 KB instruction cache.

discusses not only the number of MGT banks supported, but also the number of entries, as well as the the length and composition of the mini-graphs supported. The size of each MGT entry remains a constant 4 bytes.

The MGT's size is determined by both the number of banks and entries that it has. Also, for the mini-graph processor to be properly configured, the MGT must have enough banks to support the length of the ALU Pipeline. A mini-graph processor with a 2-stage ALU Pipeline, for example, can have anywhere from two to six banks. Six banks supports a mini-graph comprised of two integer operations, a 3-cycle load, and a conditional branch. Although technically the MGT is a cache and can support any number of mini-graph templates independent of MGT size, MGT misses are expensive [13], and therefore knowing *a priori* how many templates the MGT can hold without conflict helps performance. Because the software tool in this dissertation selects only as many templates as will fit in the MGT, the number of MGT entries determines how many mini-graph templates are supported.

Coverage. Figure 4.12 shows the configuration of every possible 8- to 512-entry MGT that supports a 2-stage ALU Pipeline. The x-axis shows MGT size in KB ($num\_entries \times banks \times sizeof(one\_bank's\ entry)$) and the y-axis shows coverage. For any given number of MGT entries, more banks usually translates to more coverage and always translates to greater MGT size. Increasing the number of banks usually offers diminishing returns; for example, for an MGT with 512 entries (black squares), the difference in coverage between 2 and 3 banks is greater than the difference between 5 and 6. This is because coverage is determined by both length and frequency; although 6-cycle mini-graphs are longer than 3-cycle mini-graphs, 3-cycle mini-graphs are far more frequent. Supporting mini-graphs of 3 cycles in length greatly increases the number of possible mini-graphs; this is not the case for 6-cycle mini-graphs.

Points with the same x value are two different configurations that have the same MGT size in KB. For example, the points labeled "6 banks, 256 entries" and "3

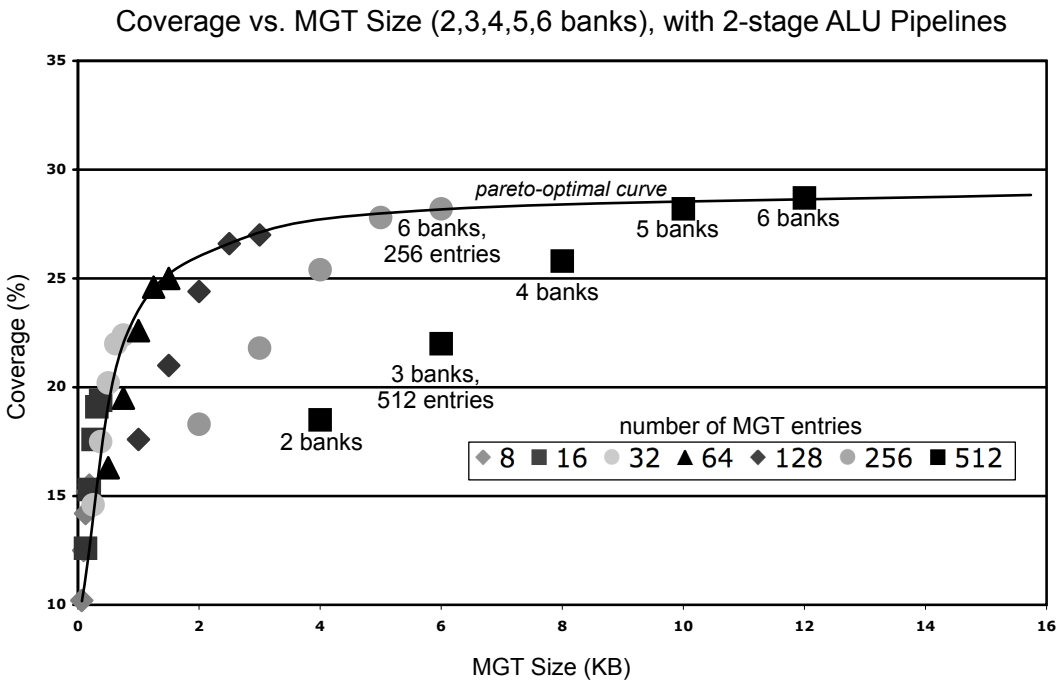


Figure 4.12: MGT Coverage vs. Size in KB

banks, 512 entries" both have 6 KB MGTs. The former has 28% coverage, the latter 22%. Assuming both configurations have comparable MGT cost, the configuration with the greater coverage if preferable. With this in mind, the graph sketches the *pareto-optimal curve*—the line along which an MGT of a particular size offers the best coverage. These configurations maximize coverage at minimal cost.

Figure 4.13 extends Figure 4.12, showing the pareto-optimal plots for 2-, 3-, and 4-stage ALU Pipelines. The graph offers two important pointers for designing a mini-graph processor. First, there is a diminishing return in supporting ever-longer ALU Pipelines. Increasing from 2- to 3-stages increases coverage by several percent. The increase from 3- to 4-stages is approximately only 1%. Depending on what the additional percent yields performance-wise, a 3-stage ALU Pipeline is likely the best cost-benefit tradeoff. Second, there are diminishing returns in supporting ever-more numbers of MGT entries. The coverage gain from 128 to 256 entries is a few

percent, and the subsequent gain in supporting 512 entries is again about a percent, depending on the number of banks.

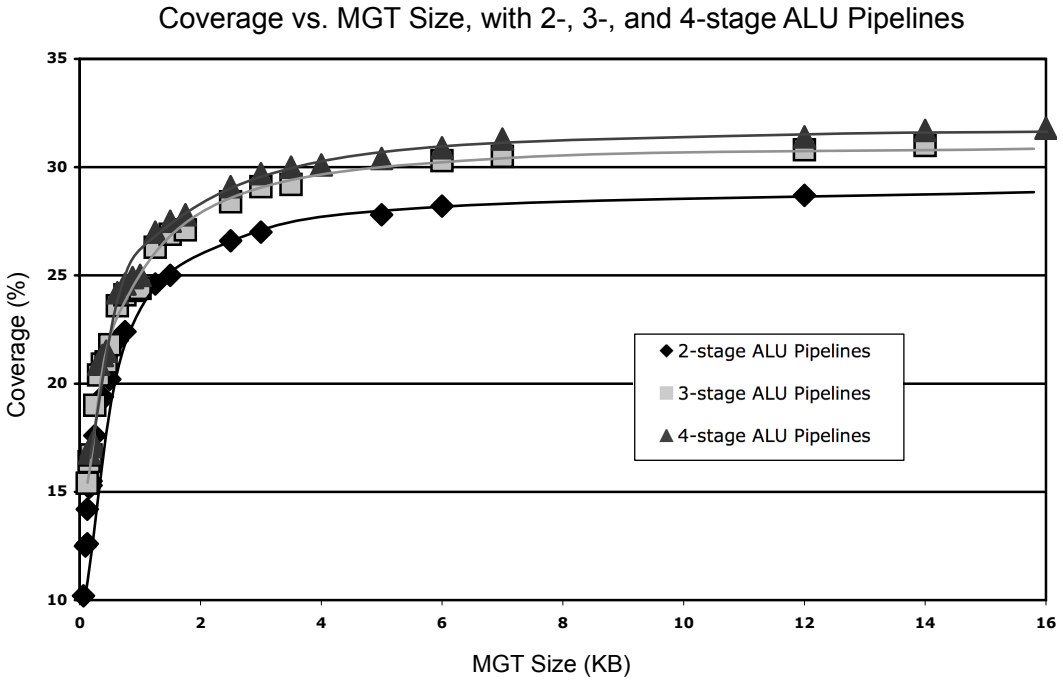


Figure 4.13: Pareto-Optimal MGT Sizes for 2-, 3-, and 4-stage ALU Pipelines

Performance. As discussed in Section 4.2, it is not always obvious how coverage translates to performance. The graph in Figure 4.14 shows performance relative to a 3-wide baseline for four different MGT configurations. These are pareto-optimal configurations with 3-stage ALU Pipelines with MGT sizes of 1, 3, 6, and 12 KB. (MGT sizes of 2 and 4 KB happen to lie just under the pareto-optimal curve.) These configurations achieve speedups of 14.4%, 16.3%, 17.2%, and 17.5%, respectively. Each curve is a unique MGT/ALU Pipeline configuration for which mini-graphs are specifically chosen: an MGT with n banks, m template entries, and p-stage ALU Pipelines.

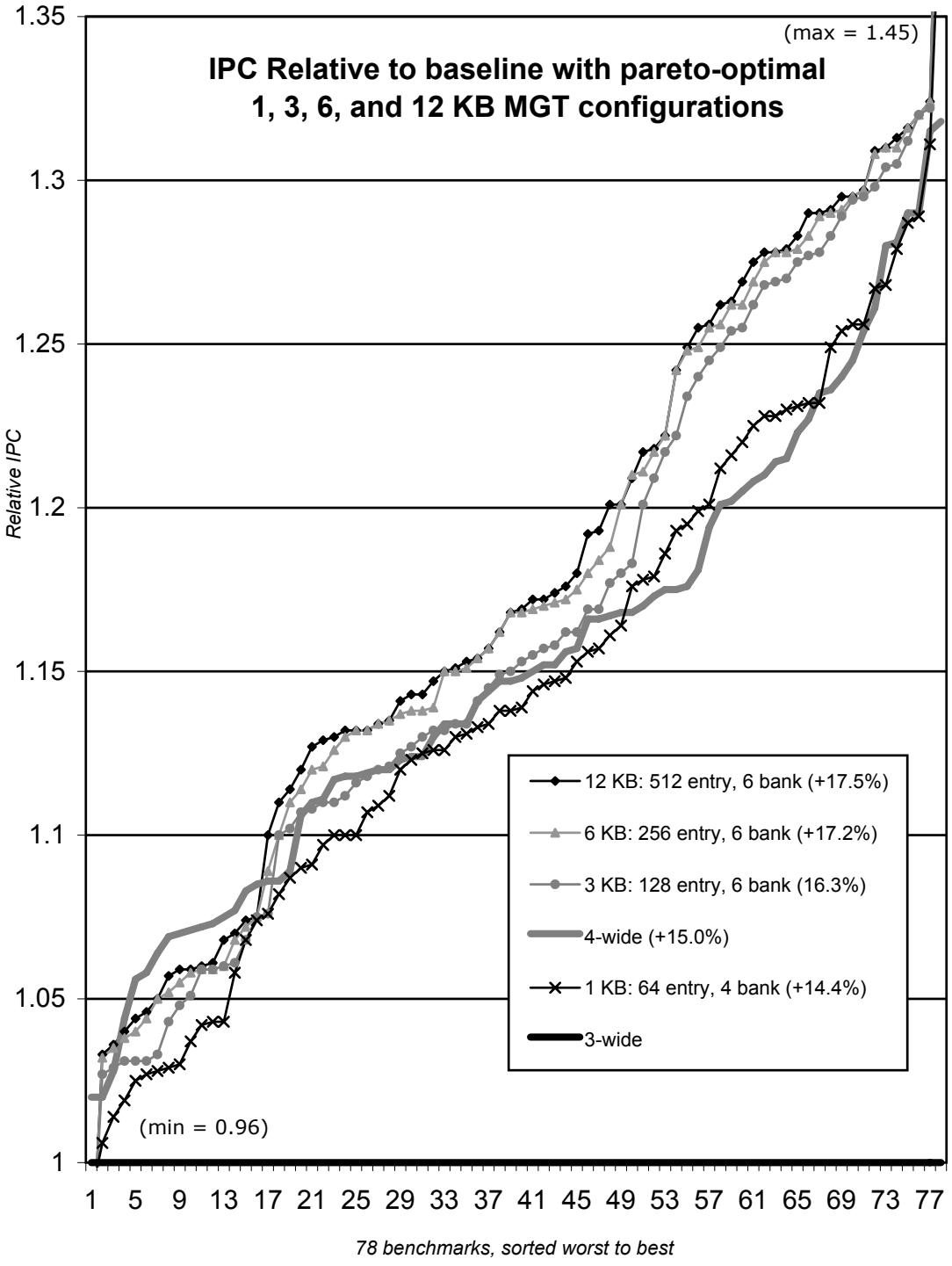


Figure 4.14: Mini-Graph performance with pareto-optimal MGT configurations of 1, 3, 6, and 12 KB in size. All support 3-stage ALU Pipelines.

For reference, the final s-curve (*bold, grey*) plots the 15% performance improvement of a 4-wide processor over the 3-wide baseline. In order to achieve this performance with a 3-wide mini-graph processor, one might assume coverage needs to be 33%. Figure 4.14 shows this assumption to be slightly conservative. The 1 KB configuration (*black x*) with only 24% coverage comes close to the 15% gain with a 14.4% improvement. The 3 KB configuration (*grey circle*) with only 29% coverage surpasses the 15% gain with a 16.3% improvement. That 33% coverage is not needed is intuitive; 33% coverage is needed only if you have sustained 4-wide ILP, which rarely happens in practice. This graph indicates that a smaller MGT in the range of 1 to 3 KB offers adequate performance. Furthermore MGT sizes past 6 KB do not offer a good return on investment; doubling the MGT from 6 KB (*grey triangle*) to 12 KB (*black diamond*) improves performance by a mere 0.3%.

Alternative, dense MGT organization. As explained in the introduction to Chapter 2, if a mini-graph does not execute a new constituent at a particular cycle, the corresponding bank's entry in the MGT remains empty. For example, a mini-graph beginning with a 3-cycle load followed by an add uses the MGT entries in banks 1 and 4 only. At cycles 2 and 3, no new constituent is executed.

While the 1-to-1 banks-to-cycle assumption of this sparse design simplifies the interface logic of the MGT, it does so at the expense of area efficiency. Every non-terminal, multi-cycle operation inside a mini-graph effectively wastes MGT entries. By adding latches to delay the output of MGT entries the appropriate number of cycles, the MGT could be packed more densely. A load-add mini-graph, for example, would read the load from the first bank of the MGT at cycle 1, read the add from the second bank of the MGT at cycle 2, and then latch the add for 2 cycles until the load completes. Mini-graphs with non-terminal loads would fit into MGTs with 2 fewer banks. This could greatly improve the coverage of MGTs with fewer banks. It has the added benefit of making the MGT configuration independent of the execution latency of mini-graph constituents.

The experiments of Section 4.4 suggest that mini-graph processing can free up to 8 KB of real-estate by exploiting instruction cache amplification to support reduced instruction cache capacity. This section shows that an 8 KB budget is indeed adequate to configure a good return-on-investment MGT. Performance gains comparable—and often superior—to those associated with increasing the superscalar width and window capacity can be achieved with an MGT requiring just 1 to 6 KB in size.

Chapter 5

Related Work

Mini-graph processing is first and foremost a form of instruction fusion that targets capacity and bandwidth amplification, although it also supports a limited form of latency reduction. Instruction fusion techniques are many in number and approach. Most, however, do *not* target amplification as mini-graphs do. Section 5.1 discusses these related fusion techniques and how they compare to mini-graph processing with regards to both objectives and execution.

Additionally, there are many non-fusion techniques that *do* target capacity and/or bandwidth amplification. These works take an entirely different approach from that of mini-graph processing, but achieve similar amplification results. These related works are discussed in Section 5.2.

5.1 Fusion Techniques

CISC/RISC. Historically, high granularity instructions have been associated with *Complex Instruction Set Computers (CISC)*. CISC instructions take many inputs, perform several operations, and produce several outputs. Although CISC programs have a low instruction count, executing CISC instructions on out-of-order processors

requires sophisticated algorithms. The decode logic supports many opcodes and multiple addressing modes; the issue stage coordinates many inputs per instruction; and execution logic supports highly-specialized, complex operations throughout which precise state is also maintained. Store-load forwarding and memory disambiguation can also be particularly difficult. The bookkeeping and coordination costs of complex instructions prevent a CISC processor from scaling efficiently. Consequently, the superscalar processors that support the x86 architecture require that the CISC instructions be broken into micro-operations.

RISC (Reduced Instruction Set Computer) instructions perform a single operation on a few register inputs and produce a single output. Although they can be executed on simpler (faster, smaller) hardware, RISC ISAs are often associated with higher dynamic instruction counts [9]. Bookkeeping expenses soar not because of the per-instruction costs but because of the large number of instructions. The hardware algorithms for instruction coordination are simpler, but the on-chip structures that support them are physically larger.

Mini-Graphs are neither CISC nor RISC instructions. They have the multiple operations and corresponding low instruction count of CISC and the simple interface and corresponding low per-instruction costs of RISC. Mini-graph processing emulates a high granularity instruction set architecture with low per-instruction bookkeeping costs. Mini-graphs reduce a program's instruction count but can be executed with simple coordination.

Micro-op fusion and macro fusion. Intel's Pentium M [38, 51] fuses load/execute and store-address/store-data micro-op pairs, reducing the number of micro-ops that are renamed, scheduled, and retired and amplifying issue queue and reorder buffer capacity. *Micro-op* fusion also reduces the number of x86 instructions that decode into multiple micro-ops, allowing the Pentium M to achieve high decoding bandwidth with a single complex decoder.

Macro fusion fuses test/branch and compare/branch pairs of macro instructions

prior to decode, similarly amplifying pipeline stages including decode and both the issue queue and reorder buffer capacity. Macro fusion further exploits latency reduction by executing these pairs on a modified branch unit that can execute both instructions in a single cycle. Simple extensions to the x86 ISA for fusing dependent instruction pairs have also been proposed [47, 48].

AMD's Athlon and Opteron processors [1] support *de facto* fusion of load-execute instructions by simply not splitting them into micro operations in the first place. Instead, these integer instructions remain as macro instructions throughout the pipeline. The instruction is renamed and issued as a single instruction, offering an amplification effect similar to micro-op fusion.

Mini-graph processing extends the benefits micro-op and macro fusion from select stages to the entire pipeline and from limited instruction pairs to more general instruction groups. Unlike the above techniques, mini-graph processing forms instruction groups offline and targets each application individually, supporting many application-specific idioms rather than a few general ones.

Macro-op scheduling and Dynamic Strands. *Macro-op scheduling* [58] temporarily and micro-architecturally fuses dependent instructions in order to boost effective scheduling capacity and hide scheduling loop latency [11, 98]. Macro-op scheduling is completely transparent, but does not amplify the bandwidths or capacities of any other structures. Whereas mini-graph processing amplifies all pipeline stages for the purpose of amplification, macro-op scheduling amplifies the scheduling stage for the sole purpose of overcoming the performance penalty of a 2-cycle pipelined scheduler. Mini-graph processing's amplification benefits can also help compensate for the penalty of a 2-cycle scheduler, but to a lesser degree because mini-graphs are more restricted that macro-ops.

Dynamic strands [89] extend macro-op scheduling beyond pair-wise fusion, dynamically fusing chains of up to three instructions and executing these chains on

closed-loop ALUs. Like macro-op scheduling—and many dynamic fusion schemes—dynamic strands can also cross basic block boundaries. Dynamic strands are speculative in two ways. First, unlike macro-op scheduling, a dynamic strand processor speculatively detects transience within a strand. Second the presence of the strand itself is speculatively detected. Whereas macro-op scheduling actively fuses instructions as the constituents are identified in the dynamic instruction stream, dynamic strands identify the beginning of a possible strand and a dispatch engine injects the entire strand into the instruction stream from the strand cache, possibly before all of the constituents are even fetched. In this respect, dynamic strands resemble data-driven multi-threading [17, 24, 86]. Mis-speculation occurs either when an assumed transient value is needed external to the strand or when an incorrect strand is dispatched. Recovery from both cases of mis-speculation rely on a hardware roll-back.

Static strands. *Static strands* [90] is the fusion technique perhaps most similar to mini-graphs. Static strands are chains of dependent instructions that, like mini-graphs, can be processed at as a single instruction at many pipeline stages. As with mini-graphs, various combinations of inputs and strand lengths are possible. Unlike mini-graphs, strands must be dependent and their focus—both in the strand processor description and coverage rates—is ALU-only sequences.

The chief difference between static strands and mini-graphs is their amplification benefits. By focusing on dependent ALU instructions, strands offer less coverage than mini-graphs. Strands also differ in the number of stages amplified. Static strands lie between mini-graphs and micro-op fusion on the pipeline amplification spectrum. This is as a result of the differences in the encoding schemes. Although both static strands and mini-graphs leverage binary compatible annotations to indicate each instance of an aggregate, mini-graph encoding uses the mini-graph handle itself as the pre-aggregate annotation. Strands are tagged, but not with a meaningful handle; their interface is generated dynamically with each occurrence. Furthermore, the strand encoding scheme performs annotation but not outlining. As a consequence,

strands do not amplify instruction cache capacity or fetch or decode bandwidth. On the contrary, the annotations in the strand binary effectively *decreases* the instruction cache capacity and the fetch and decode bandwidth. Although this does not pose a performance liability within the embedded context in which static strands are studied, initial studies indicate that this would incur a performance hit within the context of dynamically scheduled, superscalar processors in which mini-graph processing is studied [13].

Static strands do not use an MGT to store aggregate templates. Instead, a strand processor generates a compressed representation for every dynamic aggregate into a strand accumulation buffer (STAB), and then copies that compressed representation into the issue queue which is modified to support a more complex issue queue entry format. The STAB—designed to hold in-flight strands—acts as a strand ROB, whereas the MGT—designed to hold all mini-graph templates for a given binary—acts as a mini-graph instruction cache. Whereas mini-graphs execute on ALU Pipelines which inherently amplify execution bandwidth, static strands execute on closed-loop ALUs. Closed-loop ALUs double execution bandwidth only when "double-pumped."

The differences between mini-graphs and static strands illuminate the differences in their design goals. Mini-graph processing is first and foremost an amplification technique which targets dynamically scheduled superscalar processors. Static strands target power efficiency in embedded processors. That the STAB is smaller than the MGT makes it more appropriate for embedded processors. Conversely, that static strands do not technically amplify execution bandwidth and "pinch" the front of the pipeline is more acceptable for embedded processors. The IBM PowerPC [27], for example, has a fetch bandwidth twice that of its dispatch width, significantly mitigating the "pinching" effect of static strands.

CCA Graphs and MECIs. Rather than targeting amplification, *Configurable Compute Accelerator (CCA)* [19, 21, 23] graphs target performance improvement via

custom or pseudo-custom acceleration. CCA research pre-dates mini-graph processing by one year, but the two approaches are essentially concurrent. Compared to mini-graph processing, the CCA approach offers more performance improvement at a higher implementation cost. The CCA is a 4-input, 2-output functional unit as many as 7 operations deep. The CCA supports the execution of ALU instructions and simple logical operations only, forgoing the coverage benefits as well as the implementation costs of supporting memory and control instructions. Supporting only limited functionality at each CCA stage limits the levels of logic required and enables multiple operations to be executed in a single cycle.

Within a CCA binary, sequences selected to run on CCAs are delineated by two new instructions in the ISA which mark the beginning and the end of each sequence, respectively. The start instruction indicates the location of the sequence outputs and the depth of the sequence. The end instruction serves as a simple flag. As for the encoding of static strand and mini-graph binaries, these ISA extensions are assumed to be interpreted as `nop`sin the case where CCA code is run on a processor that does not support CCA.

When a rePLay pipeline [78] is not present, sequences selected to run on CCAs can be removed from the binary at decode. As with static strands, however, these sequences must be analyzed in order to generate the necessary control bits for CCA execution. This analysis introduces added latency for which CCA execution must compensate. One proposal to overcome this challenge is to store the control bits in the binary and then load them into a translation table at runtime. This solution is akin to the original DISE-based encoding scheme for mini-graphs [12]. As with the mini-graphs and the MGT, the number of subgraphs in a CCA binary is limited by the size of the translation table.

There are many variations of the CCA proposals, including both static and dynamic subgraph discovery as well as VLIW and superscalar processor targets. On a VLIW, CCA graphs consume an integer issue slot. On a superscalar, CCA graphs

consume two rename and issue slots.

Similar to the CCA approach are *Multi-Exit Custom Instructions (MECIs)* [74], which execute on a reconfigurable matrix of functional units that have the novel capability of conditional execution. By definition, mini-graphs terminate at any control instruction. MECIs, on the other hand, can contain *both* instruction paths following a branch. Both inputs to and outputs from the MECI are conditioned upon branch outcomes.

The CCA does not execute branches, and neither CCA nor MECIs perform memory operations. Both also incur reconfiguration penalties whenever a new instruction group is encountered and the configurable functional units are dynamically reconfigured accordingly. These limitations are acceptable when execution latency reductions can compensate for the reduction in issue bandwidth, particularly for workloads that are computation intensive and have fewer memory instructions in the first place. Because mini-graphs target bandwidth amplification and offer minimal latency reduction, however, limiting coverage to integer operations only and incurring regular reconfiguration penalties are untenable barriers.

Interlock-collapsing ALUs. A simple approach to latency reduction are *Interlock-collapsing ALUs (ICALUs)* [65, 82]. ICALUs can execute 2 interlocked (*i.e.*, dependent) integer operations (usually arithmetic) in a single cycle. Obviously, ICALUs are more restricted in scope than FPGA approaches. That said, they fit more seamlessly into existing processor designs, acting as a standard functional unit rather than a special purpose piece of hardware. Additionally, they incur no reconfiguration penalty. Many fusion techniques [89, 91, 112] leverage ICALUs as the functional unit of choice for accelerating the execution of two fused instructions. One such technique *not* discussed here is $RENO_{CP}$ [81], which generally falls under the category of non-fusion and is therefore discussed in Section 5.2.

Fine-grain reconfigurable accelerators. Beyond the CCA framework and ICALUs lie *fine-grain reconfigurable accelerators*. Hardware that executes more than

two fused operations often utilizes FPGAs which can be programmed to perform more arbitrary computations of, say, n operations in less than n cycles. Examples of these approaches are GARP [45], OneChip [16], PRISC [83], DISC [107], and PRISM [6]. In order to be effective, however, the speedup of the FPGA compensates for both the added time it takes to configure the unit—which may add as many as 1-2 cycles prior to each use—and the longer latency of execution; FPGA's are considerably slower than their microprocessor or ASIC counterparts. Once again, these timing penalties are invariably overcome for techniques that specifically target latency reduction in the first place, but would be infeasible for amplification techniques like mini-graph processing to overcome.

Many of the proposed techniques for reducing the execution latency of arithmetic operations target scalar in-order pipelines that can more easily accommodate multiple register inputs and outputs [45, 109, 112]. Mini-graphs, on the other hand, target dynamically-scheduled superscalar processors.

Extensible processors. In order to achieve the goal of small changes to the baseline processor and pipeline, mini-graphs conservatively execute on existing hardware. Although ALU Pipelines are technically novel, they have the same interface as existing ALUs, can process simple non-mini-graph instructions, and can be "dropped into" existing processor designs. Furthermore, they are general purpose in function and are programmed at runtime only insofar as standard ALUs are programmed. Many other forms of fusion rely on custom hardware that can execute richer instruction groups. For this, however, they have a much higher implementation cost.

There is considerable work aimed at discovering and exploiting graphs of arithmetic operations whose latency can be reduced via custom hardware. This work usually falls under the category of the (automatic) generation of application specific instruction set extensions [5, 15, 21, 23].

One commercial effort to support application specific ISA extensions is Tensilica's Xtensa [41]. As a synthesizable processor [40], Xtensa does not match the

raw frequency of traditional processors, but in the context of target applications, customization and configurability enables the Xtensa processor's high throughput to compensate for the reduced frequency, achieving speedups as high as a factor of 11. Rather than tuning the *processor* to the (high-ILP) application, mini-graph processing focuses on tuning the mini-graph templates to the application, enabling it to execute mini-graph-prepared binaries faster, while still supporting all other forms of general purpose (non-mini-graph aware) binaries. Furthermore, as a software approach, mini-graphs enable post-synthesis customization.

VLIW and EPIC. *Very Long Instruction Word (VLIW)* and its descendent *Explicitly Parallel Instruction Computing (EPIC)* offer a different form of fusion in which multiple, *independent* operations are encoded in a single instruction. This approach does not amplify processor resources as mini-graph processing does, but it does simplify both fetch and dispatch stages by placing the burden of determining independence on the compiler rather than the out-of-order engine. IBM's POWER5 [37, 93] has a similar simplification (rather than amplification) effect by dynamically grouping consecutive instructions after fetch and performing decode, dispatch, and commit *en masse*.

5.2 Non-Fusion Techniques

Capacity and bandwidth amplification. Many non-fusion techniques achieve the amplification effects similar to those of mini-graph processing but do so in alternative and/or complementary ways of mini-graph processing.

Unified renaming [54], *Instruction Reuse* [95], *Register Integration* [80], *RENO* [81], and *NoSQ* [92] all use physical register sharing and map table short-circuiting to simulate the execution of register writing instructions without actually executing them, achieving amplification in the out-of-order core, specifically register file

and issue queue capacity, and schedule, register read/write, and execution bandwidth. NoSQ targets stores as well as loads that forward from stores. RENO targets redundant loads and register-immediate add instructions. Whereas RENO and NoSQ target specific instructions, mini-graphs target *all* instructions. RENO and NoSQ amplify out-of-order execution bandwidth, particularly load and store execution bandwidth. Mini-graphs, on the other hand, amplify ROB and instruction cache capacity and the in-order front-end and back-end bandwidth. These disjoint targets—both in terms of instructions and pipeline stages—suggest that RENO and NoSQ could be synergistic with mini-graphs.

Continuous Optimization [30] and *chained in-order/out-of-order double-core architecture* [79] both prepend the out-of-order core with a simple one or two stage in-order execution engine that executes instructions whose inputs are ready. Both techniques amplify the out-of-order execution engine much like RENO and NoSQ. Their approach, however, is one of pre-execution rather than simulated execution, and as a consequence they do not amplify register file capacity or bandwidth as both RENO and NoSQ do.

Mini-graphs amplify the register file by keeping transient values alive only within the bypass network; transient values are simply never written to or read from registers. Many other techniques target a similar effect but through alternative means. *Early Physical Register Release* [29], *Physical Register Inlining* [62], and *Simple Physical Register Sharing* [103] use aggressive reclamation to amplify physical register capacity. *Virtual Physical Registers* [39] uses delayed allocation to do the same. *Checkpoint Processing and Recovery (CPR)* [2] uses a combination of aggressive reclamation and checkpointing.

The *Waiting Instruction Buffer (WIB)* [59] uses forward slicing to amplify issue queue capacity in the shadow of an L2 miss. *Continual flow pipelines (CFP)* [97] is a non-blocking architecture that uses the same technique to amplify both issue queue and register file capacity. *Checkpointed Early Resource Recycling (Cherry)* [66] uses

checkpointing and aggressive reclamation to amplify register file, ROB, and load and store queue capacities. By reordering instructions before they enter the issue buffer, *Dataflow Pre-scheduling* [68] amplifies the issue queue capacity.

Classic outlining [28, 61, 99], or "code factoring," is a compiler technique for code compression, *i.e.*, instruction cache capacity amplification. Classic outlining compresses the program binary by outlining large, unrestricted, redundant sequences of code and mapping multiple jumps to the same outlined code. Although annotated outlining actually *enlarges* a program binary—outlined code is not shared—it actually *compresses* the instruction cache footprint because the outlined code never even enters the instruction cache in the first place. Furthermore, once the outlining jump has been overwritten with a handle in the instruction cache, the mini-graph binary is no longer outlined as far as the processor is concerned. In contrast, classic outlining continues to execute all the outlined jumps as it assumes no on-chip facility for storing the outlined code.

Horizontal clustering. *Clustered Microarchitectures* divide processor resources into logical groups (or clusters) and introduce an hierarchical communication between the clusters. They improve IPC by physically amplifying processor resources that under a non-clustered organization would normally be too latency-sensitive to increase without impacting clock frequency or the pipeline. Intra-cluster communication is fast and high-bandwidth. Inter-cluster communication is slow and low-bandwidth. Ideally, instructions are steered to clusters where most of their communication will be local.

Broadly speaking, there are two forms of clustering techniques which can be distinguished by how they divide the instruction stream. *Horizontal clustering* has a single fetch stream and divides the instructions at dispatch. Commercially, the best known horizontal clustering architecture is the Alpha 21264 [44, 56], which fetches instructions and then sends them to one of two integer execution clusters, each with a distinct register file.

The chief differences between horizontal clustering techniques is whether the steering occurs statically or dynamically, and the extent to which hierarchical communication is exposed in the ISA. There are many non-commercial examples of horizontal clustering, beginning as far back as 1982, with the *Decoupled Access/Execute Architecture* [94]. In 1995, Palacharla and Smith proposed *Decoupled Integer Execution* [75], which augments often under-utilized floating-point units in order to support an optional integer execution cluster. The *MultiCluster Architecture* [31] uses a static register allocation and scheduling algorithm to balance workloads across clusters that have replicated register files, issue queues, and functional units. One of many MultiCluster descendants is RingScalar [104], a uni-directional ring of clusters with a distributed—not replicated—register file.

The *PEWs* [55] architecture performs steering dynamically, simply attempting to place each instruction in the cluster where its inputs are produced. Narayanasamy *et al.* [70] proposed a *Clustered EPIC* architecture with a 3-, 2-, and 1-wide pipeline combination as a complexity-effective alternative to a monolithic 6-wide design. Their clustered architecture executes statically identified dependency chains. *Instruction Level Distributed Processing (ILDP)* [57] consists of a common pipelined front-end, followed by a number of distributed, in-order processing elements (PEs). ILDP is an example of static clustering with ISA support for communication. Chains of dependent instructions are recognized by the microarchitecture from the instruction set itself and steered to PEs for execution. A clustered trace cache processor [10] combines clustered execution with an instruction trace cache, supporting dynamic cluster assignments optimized at retire-time.

There is extensive work on the role of efficient interconnects [77], communication latencies and pipeline depth [8], dynamic tuning [7], and steering policies [88] in increasing the performance and efficiency of clustered architectures.

Finally, one popular cluster-like approach that is currently gaining interest in the advent of multi-core research is the concept of the reconfigurable CMP. *Core Fusion*

[53] and *Federated Cores* [101], for example, both propose simple cores that can be dynamically fused into larger, high-performance, clustered cores on demand.

Vertical clustering. *Vertical clustering* has multiple fetch contexts and leverages offline steering—either in the compiler or in a trace cache. The *Multi Scalar* Architecture [34] of the early 1990's charges the compiler with splitting a program into tasks, then executing them on parallel processing elements. By performing this split prior to runtime, the compiler keeps the majority of communication on the decentralized processing elements. By identifying "hot" segments of code, trace processors [85] perform compiler optimizations dynamically. The PARROT Architecture [84] similarly finds hot traces of code and applies compiler optimizations dynamically to remove instructions, resulting in a more power-efficient processor.

There are many similarities between mini-graph processing and clustering; in most cases, the common benefits are seen at a finer granularity for mini-graphs. For example, mini-graphs leverage the locality of value communication between constituents in a mini-graphs. Transient values require neither global bypass nor to be written to the register file. The same effect is seen in clustering on a higher level; intra-cluster communication remains local to the cluster and need not be broadcast to the rest of the processor. Both techniques also effectively amplify processor resources without the clock penalties normally associated with making latency-sensitive structures larger (and consequently slower). This applies to the register file, issue queues, and bypassing logic.

Mini-graphs can achieve amplification rates of 30-40%. Horizontal clustering can theoretically achieve much higher rates, but contends with global communication delays and scale front-end bandwidth to match the amplification seen after dispatch. Ultimately, although they share benefits, mini-graphs are orthogonal to both forms of clustering; these techniques could easily be used in conjunction with one another.

Dataflow. Mini-graph processing focuses on exploiting dataflow graphs in the context of conventional ISAs and superscalar microarchitectures. Much previous

work leverages the dataflow patterns of a program holistically using new instruction sets and new microarchitectures. An entire research body on dataflow machines beginning in the 1980's [73, 87, 105] does precisely this. Several early examples include MIT's *Tagged-Token Dataflow Architecture (TTDA)* [4] and *Monsoon/Explicit Token Store (ETS)* Architecture [76]; more modern examples include *Grid Processor (TRIPS)* [69] and *WaveScalar* [67, 100]. Regardless of implementation, dataflow machines attempt to achieve more scalable single-thread execution by distributing computation and communication via direct producer-consumer communication. Mini-graphs achieve this as well, but on a much smaller scale (*i.e.*, within each mini-graph).

Chapter 6

Conclusions

Mini-graph processing is a unique form of instruction fusion that targets bandwidth and capacity amplification throughout the entire pipeline, from fetch to commit. By performing certain actions once per aggregate instead of on a per-instruction basis, structure bandwidth and capacity is allocated to other instructions, creating an amplification effect. This wholesale amplification enables either improved IPC throughput at a fixed resource point or, alternatively, fixed (or better) IPC with fewer resources. Experiments show that across four benchmark suites, the addition of mini-graph processing allows a dynamically scheduled 3-wide superscalar processor to match the IPC of a 4-wide superscalar machine.

Mini-graphs are aggregates with the external appearance of singleton RISC instructions. Mini-graphs are designed to maximize amplification both from a dynamic instruction standpoint and from the standpoint of number of structures and pipeline stages amplified. This dissertation focuses on two types of mini-graphs: *integer mini-graphs*, which contain only single-cycle ALU operations and *integer-memory mini-graphs* which can also contain, stores, loads, and conditional branches. These definitions—combined with the restriction of *atomicity*—minimize the number of pipeline stages that are explicitly mini-graph aware, changes to the ISA, and the involvement of the operating system. In other words, mini-graphs maximize their

amplification impact while minimizing their implementation costs.

A mini-graph processor requires three new structures over an existing superscalar processor. The first structure is the Mini-Graph Table (MGT), an on-chip cache for mini-graph templates. The second is the Mini-Graph Pre-Processor, a finite state machine that programs the MGT. Finally, a mini-graph processor greatly benefits from—but does not strictly require—ALU Pipelines, multi-cycle functional units on which the integer components of mini-graphs execute.

A mini-graph processor also requires that some standard superscalar structures be slightly modified. The instruction decoder recognizes the new, reserved mini-graph opcode. The scheduler of an integer-memory mini-graph processor schedules multiple cycles into the future, performing resource reservations up to the maximum number of cycles of mini-graph execution. The issue queue's entries explicitly store the output latency of mini-graph handles for writeback reservations. The register file has a single, dedicated input bus used for non-initial mini-graph inputs. Functional units on which mini-graphs may execute receive control signals and immediate values from the MGT; they also have latches for both non-initial mini-graph inputs as well as mini-graph internal inputs.

That said, the vast majority of a mini-graph processor—the instruction and data caches, the branch predictor, the floating-point units, as well as more complicated entities such as the decoder, the load/store queue, register renaming, as well as the register and memory schedulers—is identical to its corresponding non-mini-graph counterpart.

The benefit of mini-graph processing is proportional to its *dynamic coverage*, the percent of dynamic instructions embedded into mini-graphs. Coverage measures the resource amplification introduced by mini-graph processing. Mini-graph coverage rates across 78 benchmarks is consistently above 30% across several mini-graph criteria and selection algorithms. Structures and pipeline stages which process handles are amplified—effectively one-third larger. This is achieved without physically

increasing capacity or bandwidths.

Once potential performance problem associated with mini-graph processing is called *serialization*, the introduction of an artificial dependence between two instructions by virtue of being placed in a mini-graph. This dependence can delay the execution of instructions found within a mini-graph, making them execute later than they normally do in singleton (*i.e.*, non-mini-graph) form. Serialization can degrade IPC, even to the point of overwhelming the benefits of mini-graph processing.

Although serialization is common, harmful (*i.e.*, performance-degrading) serialization is the exception. This dissertation develops three serialization-aware selection schemes that identify and reject mini-graphs with *harmful serialization only* in order to maintain high coverage rates and robust performance. The most effective serialization-aware selection algorithm, $\text{Slack}_{\text{Profile}}$, uses local slack profiles to reject mini-graphs whose estimated delay cannot be absorbed by the rest of the program. $\text{Slack}_{\text{Profile}}$ virtually eliminates serialization-induced slowdowns while maintaining high amplification rates.

Given the prevalence of the x86 instruction set, the next logical step in the development of the mini-graph processor is a thorough investigation of mini-graph processing within this realm. As discussed in Section 2.7, mini-graph processing is possible at both the macro-op and micro-op level. The increasing number of instances of non-mini-graph micro-op and macro-op fusion techniques bodes well for the utility of mini-graph processing for the x86 processor.

The mini-graph processor arrives at the computer architecture scene during a time when single-thread performance is critically important, but subject to increasingly strict area and power budgets. Classic performance-improvement techniques (increasing frequencies, deep pipelining, instruction window scaling, *etc.*) introduce new power concerns and/or show diminishing returns on investment. As industry targets multi-core designs for the foreseeable future, the performance efficiency

of the individual core will become a matter of paramount importance. Supporting mini-graph processing allows a narrower core with a smaller window to provide the single-thread performance that would otherwise be achieved with a wider-issue, larger-instruction window superscalar processor. As such, the mini-graph processor is a good candidate for contemporary processor designs.

Bibliography

[1] Advanced Micro Devices, Inc. *Software Optimization Guide for AMD64 Processors*, Sept. 2005.

[2] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proc. 36th Int'l Symposium on Microarchitecture*, Dec. 2003.

[3] D. Albonesi. Selective Cache Ways: On Demand Cache Resource Allocation. In *Proc. 32nd Int'l Symposium on Microarchitecture*, pages 248–259, Nov. 1999.

[4] Arvind and R. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3), Mar. 1990.

[5] K. Atasu, L. Pozzi, and P. Ienne. Automatic Application Specific Instruction Set Extensions Under Microarchitectural Constraints. In *Proc. 40th Design Automation Conference*, Jun. 2003.

[6] P. Athanas and H. Silverman. Processor Reconfiguration Through Instruction Set Metamorphosis. *IEEE Computer*, Mar. 1993.

[7] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Dynamically Managing the Communication-Parallelism Trade-off in Future Clustered Processors. In *Proc. 30th Int'l Symposium on Computer Architecture*, pages 275–286, Jun. 2003.

[8] A. Baniasadi and A. Moshovos. Instruction Distribution Heuristics for Quad-Cluster, Dynamically-Scheduled, Superscalar Processors. In *Proc. 33rd Int'l Symposium on Microarchitecture*, pages 337–347, Dec. 2000.

[9] Dileep Bhandarkar and Douglas W. Clark. Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization. In *Proc. Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pages 310–319, Oct. 2002.

[10] R. Bhargava and L. John. Improving dynamic cluster assignment for clustered trace cache processors. In *In Proceedings of the International Symposium on Computer Architecture*, June 2003.

[11] E. Borch, E. Tune, S. Manne, and J. Emer. Loose Loops Sink Chips. In *Proc. 8th Int'l Symposium on High Performance Computer Architecture*, Jan. 2002.

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

[13] A. Bracy and A. Roth. Encoding Mini-Graphs with Handle Prefix Outlining. Technical Report TR-CIS-06-11, University of Pennsylvania, Aug. 2006.

[14] A. Bracy and A. Roth. Serialization Aware Mini-Graphs. In *Proceedings of the 39th International Symposium on Microarchitecture*, Dec. 2006.

[15] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. Instruction Generation and Regularity Extraction for Reconfigurable Processors. In *Proc. Int'l Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Oct. 2002.

[16] J. E. Carrillo and P. Chow. The effect of reconfigurable units in superscalar processors. In *In Proceedings of the 2001 ACM/SIGDA FPGA*, pages 141–150. ACM Press, Feb. 2001.

[17] R.S. Chappell, J. Stark, S.P. Kim, S.K. Reinhardt, and Y.N. Patt. Simultaneous Subordinate Microthreading (SSMT). In *Proc. 26th Int'l Symposium on Computer Architecture*, May 1999.

[18] G. Chrysos and J. Emer. Memory Dependence Prediction using Store Sets. In *Proc. 25th Int'l Symposium on Computer Architecture*, pages 142–153, Jun. 1998.

[19] N. Clark. *Customizing the Compuation Capabilities of Microprocessors*. PhD thesis, The University of Michigan, 2007.

[20] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles, and K. Flautner. An architecture framework for transparent instruction set customization in embedded processors. In *Proc. 32nd Int'l Symposium on Computer Architecture*, June 2005.

[21] 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 Int'l Symposium on Microarchitecture*, pages 30–40, Dec. 2004.

[22] N. Clark, W. Tang, and S. Mahlke. Automatically generating custom instruction set extensions. In *Proc. 36th Int'l Symposium on Microarchitecture*, pages 94–101, Nov. 2002.

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

[24] J. Collins, D. Tullsen, H. Wang, and J. Shen. Dynamic Speculative Precomputation. In *Proc. 34th Int'l Symposium on Microrchitecture*, pages 306–317, Dec. 2001.

[25] M. Corliss, E Lewis, and A. Roth. A DISE Implementation of Dynamic Code Decompression. In *Proc. 2003 Conference on Languages Compilers and Tools for Embedded Systems*, pages 232–243, Jun. 2003.

[26] M. Corliss, A. Roth, and E. Lewis. DISE: A Programmable Macro-Engine for Customizing Applications. In *Proc. 30th Int'l Symposium on Computer Architecture*, pages 362–373, Jun. 2003.

[27] IBM Corporation. *PowerPC 750 RISC Microprocessor Technical Summary.* http://www.ibm.com.

[28] S. Debray, W. Evans, R. Muth, and J. De Sutter. Compiler Techniques for Code Compression. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, Mar. 2000.

[29] O. Ergin, D. Balkan, D. Ponomarev, and K. Ghose. Increasing Processor Performance Through Early Register Release. In *Proc. 22nd IEEE Int'l Conference on Computer Design*, Oct. 2004.

[30] Brian Fahs, Todd Rafacz, Sanjay J. Patel, and Steven S. Lumetta. Continuous optimization. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 86–97, 2005.

[31] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. The Multicluster Architecture: Reducing Cycle Time Through Partitioning. In *Proc. 30th Int'l Symposium on Microarchitecture*, pages 149–159, Dec. 1997.

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

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

[34] M. Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin-Madison, Nov. 1993.

[35] Freescale Semiconductor. *AltiVec Technology Programming Environments Manual, Version 3*, 2006. http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPEM.pdf.

[36] P. Glaskowsky. Pentium 4 (Partially) Previewed. *Microprocessor Report*, 14(8), Aug. 2000.

[37] P. Glaskowsky. IBM Raises Curtain on Power5. *Microprocessor Report*, 17(10):13–14, Oct. 2003.

[38] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. Valentine. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal*, 7(2), 2003.

[39] A. Gonzalez, J. Gonzalez, and M. Valero. Virtual-Physical Registers. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, 1998.

[40] Ricardo E. Gonzalez. Xtensa — A Configurable and Extensible Processor. *IEEE Micro*, 20(2):60–70, /2000.

[41] D. Goodwin and D. Petkov. Automatic Generation of Application Specific Processors. In *Proc. Int'l Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Oct. 2003.

[42] 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.

[43] L. Gwenapp. Digital 21264 sets new standard. *Microprocesor Report*, 10(14), Oct. 1996.

[44] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, 10(14):11–16, Oct. 1996.

[45] 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.

[46] Hewlett-Packard Company. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, Feb. 1994.

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

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

[49] Intel Corporation. *Getting Started with SSE/SSE2 for the Intel Pentium 4 Processor.* http://cache-www.intel.com/cd/00/00/01/77/17741_getting_started.pdf.

[50] Intel Corporation. *Intel Itanium 2 Processor Hardware Developer's Manual*, July 2002.

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

[52] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, May 2007. http://developer.intel.com/design/processor/manuals/248966.pdf.

[53] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core Fusion: accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 186–197, Jun. 2007.

[54] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz. A Novel Renaming Scheme to Exploit Value Temporal Locality Through Physical Register Reuse and Unification. In *Proc. 31st Int'l Symposium on Microarchitecture*, pages 216–225, Dec. 1998.

[55] G. Kemp and M. Franklin. PEWs: A Decentralized Dynamic Scheduler for ILP Processing. In *Proc. 1996 Int'l Conference on Parallel Processing*, pages 239–246, 1996.

[56] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2), Mar./Apr. 1999.

[57] H-S. Kim and J. Smith. An Instruction Set and Microarchitecture for Instruction Level Distributed Processing. In *Proc. 29th Int'l Symposium on Computer Architecture*, pages 71–81, May 2002.

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

[59] A. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A Large, Fast Instruction Window for Tolerating Cache Misses. In *Proc. 29th Int'l Symposium on Computer Architecture*, May 2002.

[60] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proc. 30th Int'l Symposium on Microarchitecture*, Dec. 1997.

[61] 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.

[62] M.H. Lipasti, B.R. Mestan, and E. Gunadi. Physical Register Inlining. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.

[63] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press.

[64] Andrew Makhorin. *GNU Linear Programming Kit Reference Manual, Version 4.9*. Free Software Foundation, Inc., Jan. 2006.

[65] N. Malik, R. Eickemeyer, and S. Vassiliadis. Interlock Collapsing ALU for Increased Instruction-Level Parallelism. In *Proc. 25th Int'l Symposium on Microarchitecture*, Dec. 1992.

[66] J. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed Early Resource Recycling in Out-of-Order Microprocessors. In *Proc. 35th Int'l Symposium on Microarchitecture*, Nov. 2002.

[67] M. Mercaldi, S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin, and S. Eggers. Instruction Scheduling for Tiled Dataflow Architectures. In *Proc. 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, Oct. 2006.

[68] Pierre Michaud and Andre Seznec. Data-Flow Prescheduling for Large Instruction Windows in Out-of-Order Processors. In *Proc. 7th Int'l Symposium on High Performance Computer Architecture*, pages 27–36, Jan. 2001.

[69] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler. A Design Space Evaluation of Grid Processor Architectures. In *Proc. 34th Int'l Symposium on Microarchitecture*, Dec. 2001.

[70] S. Narayanasamy, H. Wang, P. Wang, J. Shen, and B. Calder. A Dependency Chain Clustered Microarchitecture. In *Proc. 19th IEEE International Parallel and Distributed Processing Symposium*, April 2005.

[71] Naveen Neelakantam, Ravi Rajwar, Suresh Srinivas, Uma Srinivasan, and Craig B. Zilles. Hardware atomicity for reliable software speculation. In *Proc. 34th Int'l Symposium on Computer Architecture*, pages 174–185, May 2007.

[72] Khang Nguyen. Preparing Applications for Intel Core Microarchitecture. *Technology @ Intel Magazine*, June 2006.

[73] R. Nikhil and Arvind. Can Dataflow Subsume von Neumann Computing? In *Proc. 16th Int'l Symposium on Computer Architecture*, pages 262–272, May 1989.

[74] H. Noori, F. Mehdipour, K. Murakami, K. Inoue, and M. Goudarzi. Generating and Executing Multi-Exit Custom Instructions for an Adaptive Extensible Processor. In *Design, Automation and Test in Europe Conference and Exposition*, pages 325–330. ACM, Apr. 2007.

[75] S. Palacharla and J. Smith. Decoupling integer execution in superscalar processors. In *Proc. 28th Annual International Symposium on Microarchitecture*, pages 285–290, Nov. 1995.

[76] G. Papadopoulos and D. Culler. Monsoon: An Explict Token-Store Architecture. In *Proc. 17th Int'l Symposium on Computer Architecture*, pages 82–91, Jul. 1990.

[77] J-M. Parcerisa and A. Gonzalez. The Synergy of Multithreading and Access/Execute Decoupling. In *Proc. 5th Int'l Symposium on High-Performance Computer Architecture*, pages 59–63, Jan. 1999.

[78] S. Patel and S. Lumetta. rePLay: a Hardware Framework for Dynamic Optimization. *IEEE Transactions on Computers*, Jun. 2001.

[79] Miquel Pericas, Ruben Gonzlez, Adrian Cristal, Daniel A. Jimenez, and Mateo Valero. Chained In-Order/Out-of-Order Double Core Architecture. In *Proceedings of the 17th Intl Symposium on Computer Architecture and High Performance Computing.*, Feb. 2005.

[80] V. Petric, A. Bracy, and A. Roth. Three Extensions to Register Integration. In *Proc. 35th Int'l Symposium on Microarchitecture*, pages 37–47, Nov. 2002.

[81] V. Petric, T. Sha, and A. Roth. RENO: A Rename-Based Instruction Optimizer. In *Proc. 32nd Int'l Symposium on Computer Architecture*, pages 98–109, Jun. 2005.

[82] J. Phillips and S. Vassiliadis. High-Performance 3-1 Interlock Collapsing ALUs. *IEEE Transactions on Computers*, 1994.

[83] R. Razdan and M. Smith. A High-Performance Microarchitecture with Hardware Programmable Function Units. In *Proc. 27th Int'l Symposium on Microarchitecture*, Dec. 1994.

[84] Roni Rosner, Yoav Almog, Micha Moffie, Naftali Schwartz, and Avi Mendelson. Power awareness through selective dynamically optimized traces. In *Proc. 31st Int'l Symposium on Computer Architecture*, 2004.

[85] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace Processors. In *Proc. 30th Int'l Symposium on Microarchitecture*, pages 138–148, Dec. 1997.

[86] A. Roth and G. Sohi. Speculative Data-Driven Multithreading. In *Proc. 7th Int'l Symposium on High-Performance Computer Architecture*, pages 37–48, Jan. 2001.

[87] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. In *Proc. 16th Annual Int'l Symposium on Computer Architecture*, pages 46–53, May 1989.

[88] Pierre Salverda and Craig B. Zilles. A Criticality Analysis of Clustering in Superscalar Processors. In *Proc. 38th Annual International Symposium on Microarchitecture*, pages 55–66, Nov. 2005.

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

[90] 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.

[91] Y. Sazeides, S. Vassiliadis, and J. Smith. The Performance Potential of Data Dependence Speculation and Collapsing. In *Proc. 29th Int'l Symposium on Microarchitecture*, Dec. 1996.

[92] T. Sha, M. Martin, and A. Roth. NoSQ: Store-Load Communication without a Store Queue. In *Proc. 39th Int'l Symposium on Microarchitecture*, Dec. 2006.

[93] B. Sinharoy, R. Kalla, J. Tendler, R. Eickemeyer, and J. Joyner. POWER5 System Microarchitecture. *IBM Journal of Research and Development*, 49(4/5), May 2005.

[94] J. Smith. Decoupled Access/Execute Computer Architecture. In *Proc. 9th Int'l Symposium on Computer Architecture*, Jul. 1982.

[95] A. Sodani and G. Sohi. Dynamic Instruction Reuse. In *Proc. 24th Int'l Symposium on Computer Architecture*, Jun 1997.

[96] 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.

[97] S. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual Flow Pipelines. In *Proc. 11th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[98] J. Stark, M. Brown, and Y. Patt. On Pipelining Dynamic Instruction Scheduling Logic. In *Proc. 33rd Int'l Symposium on Microarchitecture*, Dec. 2000.

[99] B. De Sutter, H. Vandierendonck, B. DeBus, and K. DeBosschere. On the side-effects of code abstraction. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 244–253, Jun. 2003.

[100] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *Proc. 36th Int'l Symposium on Microarchitecture*, Dec. 2003.

[101] D. Tarjan, M. Boyer, and K. Skadron. Federation: Out-of-Order Execution using Simple In-Order Cores. Technical Report CS-2007-11, University of Virginia Department of Computer Science, Aug. 2007.

[102] X. Tian, E. Su, D. Kreitzer, H. Saito, R. Krishnaiyer, A. Kanhere, J. Ng, C. Lim, and S. Ghosh. Inside the Intel 10.1 Compilers: New Threadizer and New Vectorizer for Intel Core2 Processors. *Intel Technology Journal*, 11(4), 2007.

[103] L. Tran, N. Nelson, F. Ngai, S. Dropsho, and M. Huang. Dynamically Reducing Pressure on the Physical Register File through Simple Register Sharing. In *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software*, 2004.

[104] Jessica H. Tseng and Krste Asanovic. RingScalar: A Complexity-Effective Out-of-Order Superscalar Microarchitecture. Technical Report MIT-CSAIL-TR-2006-066, Massachusetts Institute of Technology, CSAIL, Sept. 2006.

[105] A. Veen. Dataflow Machine Architecture. *ACM Computing Surveys*, 18(4):365–396, Dec. 1986.

[106] Ofri Wechsler. Inside Intel Core Microarchitecture: Setting New Standards for Energy-Efficient Performance. *Technology @ Intel Magazine*, March 2006.

[107] M. J. Wirthlin and B. L. Hutchings. DISC: The dynamic instruction set computer. In *In Field Programmable Gate Arrays for Fast Board Development and Reconfigurable Computing*, pages 92–103, 1995.

[108] 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.

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

[110] K. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, Apr. 1996.

[111] S. Yehia, N. Clark, S. Mahlke, and K. Flautner. Exploring the design space of LUT-based transparent accelerators. In *Proceedings of the 2005 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 159–170, Sept. 2005.

[112] 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 Int'l Symposium on Computer Architecture*, pages 159–170, Jun. 2004.