Appears in Proceedings of 37th International Symposium on Microarchitecture (MICRO-37), Dec. 4–8, 2004.

Dataflow Mini-Graphs: Amplifying Superscalar Capacity and Bandwidth

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

Abstract

*A **mini-graph** is a dataflow graph that has an arbitrary internal size and shape but the interface of a singleton instruction: two register inputs, one register output, a maximum of one memory operation, and a maximum of one (terminal) control transfer.*

Previous work has exploited dataflow sub-graphs whose execution latency can be reduced via programmable FPGA-style hardware. In this paper we show that mini-graphs can improve performance by amplifying the bandwidths of a superscalar processor's stages and the capacities of many of its structures without custom latency-reduction hardware. Amplification is achieved because the processor deals with a complete mini-graph via a single quasi-instruction, the handle. By constraining mini-graph structure and forcing handles to behave as much like singleton instructions as possible, the number and scope of the modifications over a conventional superscalar microarchitecture is kept to a minimum.

This paper describes mini-graphs, a simple algorithm for extracting them from basic block frequency profiles, and a microarchitecture for exploiting them. Cycle-level simulation of several benchmark suites shows that mini-graphs can provide average performance gains of 2–12% over an aggressive baseline, with peak gains exceeding 40%. Alternatively, they can compensate for substantial reductions in register file and scheduler size, and in pipeline bandwidth.

1. Introduction

Processors are good at executing simple instructions with small, fixed interfaces: two inputs, one output, a maximum of one memory reference, a maximum of one control transfer. Machinery for dealing with small, fixed interfaces is well understood and (relatively) easy to build. Unfortunately, because instructions are fine grained, they are also numerous. While instruction processing machinery—most of which performs inter-instruction book-keeping—may be conceptually simple, it may become physically complex by virtue of its capacity and bandwidth.

In this paper, we propose a mechanism that allows simple, fixed-interface, single-instruction machinery to process multi-instruction dataflow graphs which we call ***mini-graphs***. A mini-graph is a connected instruction dataflow graph that has the interface of a singleton instruction: two inputs, one output, at most one memory reference, and at most one control transfer. A binary rewriting tool modifies executables and statically replaces dataflow graphs that satisfy mini-graph criteria with ***handles***; a handle is a quasi-instruction that encodes the corresponding mini-graph's interface register dependences.

A mini-graph pipeline processes both unmodified and modified executables and treats handles as individual instructions in all stages except execution. During execution, the processor consults a handle-to-instruction sequence translation which is stored in an on-chip table called the ***mini-graph table (MGT)***. Essentially a microcode store, the MGT drives the cycle-by-cycle execution of the constituent mini-graph instructions with low overhead. The MGT may be hardwired, but it is more useful to customize its contents to an application. We show that ***DISE (dynamic instruction stream editor)*** is a good match for specifying application-specific mini-graphs.

Dataflow aggregates are not a new idea, but a mini-graph processor exploits them in a new way. Most previous schemes ***reduce the execution-latency*** of aggregates using custom hardware. A mini-graph processor can do that too, but primarily it ***amplifies the bandwidth and capacity*** of book-keeping machinery. A key to amplification is to constrain mini-graph structure such that handles look and behave like singleton instructions, *e.g.*, renaming a handle has the same effect as renaming each mini-graph instruction individually. This approach maximizes the number of stages (structures) that can process (store) handles rather than mini-graph instructions and whose bandwidth (capacity) is amplified.

The most important aspect of making handles behave like instructions is choosing mini-graphs that are atomic. This restriction admittedly reduces mini-graph "coverage", but allows us to treat values on a mini-graph's interior—we use static analysis to identify these values—as transient and to avoid allocating physical register storage for them. This approach reduces register file size requirements and amplifies renaming, scheduling, register read, register write, and retirement bandwidths. Since mini-graphs naturally amplify fetch bandwidth and instruction cache capacity, execution remains the only un-amplified stage. To prevent it from becoming a bottleneck, we introduce a microarchitectural component called an ***ALU pipeline***—a single-entry, single-exit chain of ALUs—which adds execution bandwidth without increasing bypassing complexity.

Execution-driven simulations of SPEC2000, Media-

Bench, CommBench, and MiBench programs show that mini-graphs produce average performance improvements of 2%, 12%, 6% and 7% respectively, over an aggressive baseline and without any latency reduction. On a per application basis, gains can exceed 30% and 40%. Mini-graphs can also effectively compensate for dramatic reductions in the capacities of the scheduler and register file and bandwidth at all pipeline stages.

We make four main contributions:

- First, we observe that instruction aggregates that have external interfaces of singleton instructions can improve performance by amplifying processor capacity and bandwidth, without requiring custom hardware for reducing dataflow-graph latency. We call such aggregates ***mini-graphs***.
- Second, we describe a microarchitecture for processing mini-graphs that requires only small modifications over existing superscalar designs.
- Third, we demonstrate that DISE is suitable for creating and using application-specific mini-graphs.
- Finally, we present a simulation-driven performance evaluation of our complete system.

2. Related Work

There is considerable prior work on the (automatic) generation of application specific instruction set extensions [1, 4, 6, 7], including commercial efforts like Tensilica's Xtensa [9]. This work has been aimed at discovering and exploiting graphs of arithmetic operations whose latency can be reduced via custom hardware. This hardware ranges in implementation from collapsing ALU [20, 22, 27] to FPGA [2, 11, 21] and in interface from functional unit [20, 21, 26, 27] to co-processor [11, 26]. Mini-graph processors *can* exploit custom hardware to reduce graph latency, but they improve performance primarily by reducing book-keeping overhead and amplifying the capacity of structures like the scheduler and register file and the bandwidth of all pipeline stages. Mini-graph interfaces—*e.g.*, number of allowed register inputs and outputs—and internal composition are highly constrained to minimize the number of pipeline stages that must be augmented with mini-graph awareness. Some of these constraints have been employed previously [20, 21], but again, only in the context of collapsible dataflow graphs.

The fusion of dependent instructions for capacity and bandwidth amplification is not entirely new, but existing forms are more restricted than the mechanisms we propose. Intel's Pentium M [13] fuses load/execute and store-address/store-data micro-op pairs, reducing the number of micro-ops that must be renamed, scheduled, and retired and amplifying issue queue 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. Simple extensions to the X86 ISA for fusing dependent instruction pairs have also been proposed [12].

Macro-op scheduling [14] temporarily and microarchitecturally fuses dependent instructions in order to boost effective scheduling capacity and hide scheduling loop latency [3, 23]. Macro-op scheduling is completely transparent, but does not amplify the bandwidths or capacities of any other structures.

There is extensive work on algorithms for choosing compound instructions to optimize coverage or performance under a variety of constraints [1, 4, 6, 7, 19]. Our microarchitectural focus complements that work.

Finally, our work focuses on exploiting dataflow graphs in a superscalar context. Grid Processor [17] and WaveScalar [24] exploit dataflow graphs holistically using new instruction sets and new microarchitectures.

3. Mini-graphs

We describe the physical structure of a mini-graph, the restrictions on it and the rationale behind them, and a simple, greedy algorithm for extracting mini-graphs from program profiles.

Figure 1a shows two code snippets from the program *gcc*. In each snippet, the shaded instructions comprise a mini-graph. Figure 1b shows the same snippets with each mini-graph replaced by a single instruction ***handle***. A handle is a quasi-instruction that is only meaningful to a mini-graph enabled processor. It has three components: i) a reserved opcode **mg**, ii) two input and one output register specifiers, and iii) an immediate field. The immediate field is called the ***MGID*** and is the index into an on-chip table, the ***mini-graph table (MGT)***, which contains the instruction-by-instruction mini-graph definition. Figure 1c shows the contents of an MGT for the two mini-graphs. Each MGT row contains the definition of one mini-graph template; row 12 contains the specification for the mini-graph on the left (MGID 12). Each MGT **INSN** column represents a mini-graph template instruction; the MGT shown here can represent mini-graphs of three instructions or less. Note, this MGT is logical; the organization and contents of an actual MGT are described in Section 4.1.

The three register names explicit in a handle are the mini-graph's ***interface registers*** which define its external dependences. The handle must contain these register names because they (or their renamed versions) are needed at renaming, scheduling, register read, register write, retirement, and misprediction recovery; stages where only the handle is available, not the complete mini-graph. Information which is only needed during execution—the ***interior registers*** which define mini-graph internal dataflow, as well as the opcodes and

(a)

```
addl r18,2,r18          ldl r18,24(r16)
lda r6,2,r6             ldq r2,16(r4)
s8addl r7,r0,r7         srl r2,14,r17
cmplt r18,r5,r7         bis zero,r18,r16
bne r7,0xA              and r17,1,r17
```

(b)

```
lda r6,2,r6             ldl r18,24(r16)
s8addl r7,r0,r7         mg r4,–,r17,34
mg r18, r5,r18,12       bis zero,r18,r16
```

(c)

MGT	OUT	INSN0	INSN1	INSN2
12	**0**	addl **E0**,2	cmplt M0,**E1**	bne M1,0xA
34	2	ldq 16(E0)	srl M0,14	and M1,1

FIGURE 1. Mini-graphs. (a) Code snippets from gcc. (b) Same snippets with shaded mini-graphs replaced by handles. (c) MGT contains mini-graph definitions.

immediates of the individual instructions—is not explicit in the handle; it is encoded in the MGT.

In the MGT, interface input registers are mnemonically denoted using the letter **E** and their index in the handle while interior values are denoted using the letter **M** and the mini-graph instruction that creates them. The MGT **OUT** field indicates which instruction produces the mini-graph's interface output register. Thus the first mini-graph instruction, **addl r18,2,r18** is represented in **INSN0** column of the MGT as **addl E0,2**; **E0** is the first interface register explicit in the handle, **r18**. The second instruction, **cmplt r18,r5,r7** is represented as **cmplt M0,E1** where **E1** is interface register **r5** and **M0** is output of the first mini-graph instruction. The final instruction **bne r7, 0xA** is represented as **bne M1,0xA**. That the output of the mini-graph is produced by its first instruction is denoted by a **0** in the **OUT** field.

3.1. Structural Constraints

The most important aspect of the appearance of being a single instruction is ***atomicity***. Atomicity constrains mini-graphs to reside within basic blocks, a severe restriction for programs with small blocks. Conventional multiple block constructs like superblocks and hyperblocks are not atomic as they have side exits. RePLay [18] frames, however, are atomic and others have shown that large dataflow aggregates can be mined from them [6, 27]. Mini-graphs can contain branches, but these must be terminal.

Coarse-grain atomicity allows mini-graph execution to be more efficient than conventional execution. Because partial mini-graph state is never needed, it is not necessary to allocate explicit storage (*i.e.*, physical registers) to partial, interior mini-graph results. Mini-graph interior values only live in the bypass network. In this respect they are similar to interior values of grid processor code blocks [17].

The fact that only mini-graph interface registers require physical registers allows for bandwidth amplification at four key pipeline stages: renaming (physical register allocation), register read, register write, and retirement (physical register freeing). Because interface register names must be explicit in the handle and because many pipeline stages have machinery that assumes two register inputs and/or one register output per instruction, we limit mini-graphs to two input and one output interface registers. Unlike atomicity which is fundamental, this constraint is only a practical one.

In contrast with previous work, we allow mini-graphs to include loads and stores. However, we limit the number of memory operations per mini-graph to one. This restriction removes ordering ambiguities that would result from two stores or a load and store to the same address within a mini-graph. It enables mini-graphs to be collapsed to single instructions while preserving total load/store order. Finally, it simplifies the handling of memory exceptions, *e.g.*, page faults.

3.2. Mini-Graph Selection

Our focus is on micro-architectural techniques for exploiting mini-graphs. We are less concerned with developing new mini-graph selection algorithms and defer to prior work in that respect [7, 19]. In this work, we use a simple greedy selection algorithm.

First, we analyze the static executable and enumerate 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. Mini-graph legality testing is more involved than simply testing the interface (two register inputs, one register output) and composition (one memory operation) conditions. The instructions in a mini-graph are not necessarily contiguous in the original program and execution semantics must not change when they are collapsed to a single handle. For each mini-graph, we choose an ***anchor*** around which to collapse the remaining instructions. In order of preference, the anchor is: i) the branch, ii) the memory operation, or iii) the last instruction. Notice, in Figure 1b the mini-graphs are collapsed around the branch and load, respectively. Memory operations are given precedence so that collapsing does not result in load/store reordering. We reject mini-graphs if there is register interference in the range between the anchor and original positions of the first and last instructions. Our static choice of anchor forces us to reject some legal mini-graphs, but our experiments indicate that this is rare.

Next, we sort the mini-graph list in order of decreasing benefit. Since our focus is on amplifying bandwidth and capacity, our mini-graph benefit function is its ***coverage***: the fraction of dynamic instructions it removes from the pipeline. A mini-graph's estimated coverage is *(n-1)*f* where *n* is its size in instructions and *f* is its execution frequency, the sum of the execution frequencies of all of its static instances (we consider static mini-graphs with identical dataflows and immediate operands as equivalent and coalesce them). We derive *f* from a basic-block frequency profile.

Finally, we select mini-graphs by iterating over the sorted candidate list. Because a single static instruction may belong to at most one mini-graph, the selection of a mini-graph may eliminate some remaining candidates. At the end of each iteration, we adjust the weights of the remaining mini-graphs. The process repeats until the list is exhausted or a preset mini-graph limit is reached.

4. Mini-graph Execution

In this section we describe a mini-graph microarchitecture. For now, we assume that the processor supports a hardwired set of mini-graphs and that the only mini-graph handles that appear in programs are legal ones.

4.1. Basic Microarchitecture

A mini-graph processor treats handles as singleton instructions at all stages except execution. There, it uses a table to control—microcode-style—the execution of the individual mini-graph instructions.

The mini-graph table (MGT). The central component of a mini-graph execution core is the mini-graph table (MGT) which maps handle MGIDs to mini-graph definitions. We have already introduced the MGT logically; now we define its physical organization. Because some aspects of a mini-graph's definition are needed during scheduling and others are needed at execution, the MGT is organized as two tables, shown in Figure 2 (there are two alternative templates for mini-graph 34).

The ***mini-graph header table (MGHT)*** contains scheduling information which includes the functional units needed by the mini-graph (**FU0** and **FUBMP**), and the latency of the its register output (**LAT**) which is shorter than the execution latency of the complete mini-graph if the output is not produced by the last instruction. Note, **FUBMP** is used for a particular style of mini-graph scheduler which we present in Section 4.3. The header allows the scheduler to reserve functional units, bypass paths, and register write ports.

The ***mini-graph sequencing table (MGST)*** is used during execution and includes the execution information for each instruction in the mini-graph. This includes: functional unit (**FU**), opcode (**OP**), immediate value (**IM**), and bypassing directives (**B0** and **B1**). The MGST is sliced vertically, one bank per mini-graph execution cycle. Integer mini-graph instructions are arranged in consecutive banks, but multi-cycle operations like loads require that subsequent banks be left empty. Note, mini-graph 34 whose first instruction is a load (here we assume that load latency is 2 cycles). MGST bank 1 is empty; the rest of the mini-graph resumes in bank 2. The rationale for this organization will be clear shortly.

We explain the structure and usage of these two tables using an example execution of mini-graph 12. In subsequent sections, we use the term MGT to refer to the MGHT and MGST collectively.

Mini-graph life cycle. Figure 3a shows a mini-graph handle as it progresses through the nominal stages of a superscalar pipeline. The bold number at the beginning of each stage action is the cycle at which the action takes place (we assume all stages are single-cycle).

A handle is fetched, decoded, and renamed as if it were a singleton instruction (the physical register in parentheses is the overwritten output register which must be freed when the handle retires). The handle is allocated reorder buffer and scheduler (reservation station, issue queue) entries. Its MGID is used to read the MGHT and both MGID and the contents of the MGHT entry are copied to the scheduler entry, the latter to avoid MGHT lookups during scheduling. The functional unit (**FU0**) required by mini-graph 12 is **AP** (ALU pipeline), a new unit we describe in Section 4.2. The latency of the register output (**LAT**) is 1 since the first instruction in the mini-graph produces the output.

The MGST is coupled to *M* pipelined sequencers, where *M* is the maximum number of handles that can be scheduled per cycle. When the handle is issued, the scheduler sends the MGID (12) to a free sequencer. Over the next three cycles, the sequencer advances from one MGST bank to the next, reading and driving the control signals for each successive mini-graph instruc-

FIGURE 2. Mini-graph table. Physical MGT organization/contents for two example mini-graphs, MGID 12 and 34.

	MGHT			MGST.0					MGST.1					MGST.2					MGST.3				
	LAT	FU0	FUBMP	FU	OP	IM	B0	B1	FU	OP	IM	B0	B1	FU	OP	IM	B0	B1	FU	OP	IM	B0	B1
12	1	AP	–:–:–	AP.0	addl	2	E0	IM	AP.1	cmplt		M0	E1	AP.2	bne	0xA	M1	IM					
34	4	LD	–:ALU:ALU	LD	ldq	16	E0	IM						ALU	srl	14	LD	IM	ALU	and	1	ALU	IM
34	4	LD	–:AP:–	LD	ldq	16	E0	IM						AP.0	srl	14	E0	IM	AP.1	and	1	M0	IM

(a)

Fetch/Decode	Rename/Alloc	Schedule	RegRead	Execute	RegWrite	Retire/Free
1: mg r18,r5,r18,12	**2:** mg p20,p14,p32(p40),1, rob,rs,preg	**3:** MGST[12]	**5:** p20,p14	**6:** MGST.0[12]	**7:** p32	
				7: MGST.1[12]		
				8: MGST.2[12]		**10:** p40

(b)

Fetch/Decode	Rename/Alloc	Schedule	RegRead	Execute	RegWrite	Retire/Free
1: addl r18,2,r18	**2:** addl p20,2,p32(p40), rob,rs,preg	**3:** addl	**4:** p20	**5:** addl	**6:** p32	**7:** p40
1: cmplt r18,r5,r7	**2:** cmplt p32,p14,p17(p15), rob,rs,preg	**4:** cmplt	**5:** p14	**6:** cmplt	**7:** p17	**8:** p15
1: bne r7,0xA	**2:** bne p17, 0xA, rob,rs	**5:** bne	**6:**	**7:** bne	**8:**	**9:**

FIGURE 3. Mini-graph execution example. (a) Mini-graph 12 executing as a handle. (b) Mini-graph 12 executing as 3 conventional instructions.

tion. The combination of MGST and sequencers act like pipelined, table-driven microcode. The MGST's cycle-based organization avoids sequencer and bank conflicts.

On the mini-graph terminal instruction, the MGST sequencer writes a completion bit in the handle's reorder buffer entry and frees its scheduler entry. Register write back and tag broadcast are handled by the scheduler which has reserved ports and tag buses for them.

A handle is retired like a singleton instruction, it overwrites and must free at most one physical register. For mini-graph containing a store, the corresponding store queue entry (of which there can be only one) is written to the data cache.

Summary. In general, mini-graphs only require changes to the scheduling and execution stages of the pipeline. Some other stages appear to require modification, but these are natural as whatever action was previously performed on a singleton instruction is now performed on the handle instead. For instance, if a mini-graph terminates in a branch, the handle PC stands in for the branch PC for the purposes of branch prediction and update. The fact that mini-graph interior values are transient and mini-graphs contain at most one memory operation makes it easy to handle mini-graph exceptions. Exception information is attached to the handle, the entire mini-graph is flushed, the exception is handled, and the entire mini-graph is replayed.

Performance effects. Figure 3b shows mini-graph 12 executing as three singleton instructions. The advantage of mini-graph execution is obvious from the difference in resource and bandwidth consumption. The mini-graph requires one slot each of fetch, decode, rename, schedule, and retire. Individual execution requires three slots at each stage. The mini-graph requires one reorder buffer entry and one scheduler entry, individual execution requires three each. Mini-graph execution requires one physical register and one physical register write, individual execution requires two registers and two writes.

The price of bandwidth and capacity amplification is the potential for two forms of serialization. ***External serialization*** potentially delays issue for mini-graphs with external inputs to instructions other than the first. Our example mini-graph 12 suffers from potential external serialization, which is illustrated in Figure 3a and 3b. Consider that registers **p20** and **p14** are ready in cycles 4 and 5, respectively. Executing individually (Figure 3b), **addl** and **cmplt** (which depends on it) execute in cycles 5 and 6, respectively. Executing as a mini-graph (Figure 3a), **addl** is spuriously forced to wait for **p14** and the two instructions execute in cycles 6 and 7.

Internal serialization produces execution delays for mini-graphs with internal parallelism, *e.g.*, a three-instruction mini-graph whose first two instructions feed the third but are independent of one another. Executing individually, this sequence could execute in 2 cycles; as a mini-graph it executes in 3. Unlike external serialization, internal serialization is not a fundamental problem; We could allow the MGST to drive the execution of two mini-graph instructions per cycle. However, internally parallel mini-graphs which expose this problem are rare and do not justify this added complexity. We investigate the cost of both forms of serialization in Section 6.2.

4.2. ALU Pipeline

A mini-graph processor executes mini-graphs composed entirely of single-cycle integer operations on an ***ALU pipeline***: a single-entry single-exit pipelined chain of ALUs. An ALU pipeline is simple because to a scheduler, it looks like a pipelined, multi-cycle functional unit, *e.g.*, a multiplier. It is powerful because it amplifies execution bandwidth to match the amplification of all other bandwidths that mini-graphs provide. It does so without adding bypass or register file complexity. A 3-stage ALU pipeline can perform 3 operations per cycle, but has only 1 register/bypass output and 2 inputs, rather than 3 outputs and 6 inputs.

Figure 4 shows the basic design: a chain of ALUs with two external inputs and a single output. The external register inputs and the outputs of each stage ALU are latched to form a pipeline. The figure shows a 3-stage ALU pipeline. The external inputs to the pipeline are for register values. Each stage ALU also has a ***side***

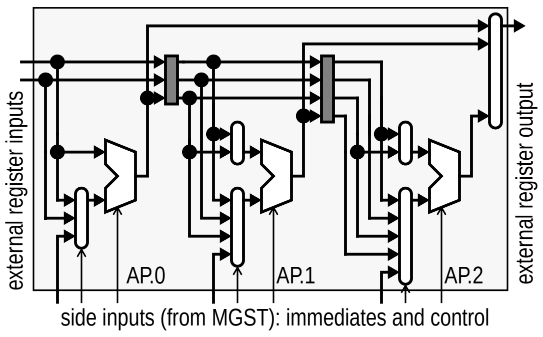


FIGURE 4. ALU pipeline. 3-stage non-collapsing ALU pipeline with no support for branches.

input. This input is an immediate which is streamed to the ALU by the MGST. An ALU pipeline need not include all possible forwarding paths; mini-graphs that require excluded paths are simply disallowed. Support for branches (not shown) requires the PC as an implicit input to the pipeline and some additional control logic.

The output of an ALU pipeline is selected between the ***unlatched*** outputs of each of the stage ALUs. This arrangement has several advantages. It doesn't penalize mini-graphs whose output register is not produced by the last instruction. It also allows us to handle long mini-graphs while not penalizing shorter ones and even execute singleton ALU operations on ALU pipelines with no penalty. This is important because it lets us substitute ALU pipelines for ALUs without complicating the scheduler or degrading the performance of programs that do not exploit mini-graphs. One issue with this design is the possibility of "writeback" conflicts; these are avoided by the scheduler with the help of header information, specifically output latency (**LAT**).

Latency-reducing ALUs. A mini-graph processor does not rely on latency reducing ALUs [16, 21, 22, 27], but can exploit them. Support for such ALUs depends on the precise manner in which latency reduction is achieved. Structured latency reduction—*e.g.*, by fusion of consecutive ALU pipeline stages using techniques like carry-save addition—is easy to incorporate into our scheme. We simply expand the MGST to allow each slice to emit control signals for two instructions. Incorporating ad hoc latency reduction hardware like an FPGA [21] is more difficult. It potentially requires an additional step prior to execution to program the unit.

4.3. Sliding-window Scheduler

Integer mini-graphs provide limited coverage. For many applications, better coverage can be achieved using ***integer-memory mini-graphs*** which can contain loads and stores. It is impractical to create a load/store pipeline or to incorporate load/store stages into an ALU pipeline. Integer-memory mini-graphs execute on a combination of conventional functional units and any ALU pipelines that exist. Their interior values live in the bypass network. In this section, we introduce a modified scheduler called a ***sliding-window scheduler*** that can schedule integer-memory mini-graphs. Unlike an ALU pipeline, a sliding-window scheduler does not amplify execution bandwidth because load and store ports are not replicated.

Basic operation. A sliding-window scheduler needs one piece of new functionality: the ability to reserve all the functional units a mini-graph will use at once. Conventional schedulers already have some forward reservation functionality which they use to reserve register write ports for multi-cycle operations. Logically, a scheduler maintains a two-dimensional reservation bitmap: one dimension represents resources (here register ports), the other future cycles. The number of future cycles represented is equal to the latency of the longest common operation, *e.g.*, load. Each cycle, the issuing instructions reserve register write ports by setting bits in the appropriate subsequent bitmap lines. Each cycle, the bitmap advances by one line. A sliding window scheduler extends the bitmap in the resource dimension to include functional units and in the time dimension to the maximum mini-graph execution latency.

To help in making mass functional unit reservations, we augment the MGHT with an **FUBMP** field that represents the functional units used by the second and subsequent mini-graph instructions; the unit needed by the first instruction is represented in the field **FU0**. The register write port bitmap is implicitly represented in the **LAT** field. In Figure 2, mini-graph 12 is an integer mini-graph: it executes on an ALU pipeline and its **FUBMP** is empty. Mini-graph 34, however, is an integer-memory mini-graph and its **FUBMP** indicates that it needs ALUs in the third and fourth cycles after issue.

Like a conventional scheduler, a sliding window scheduler initially schedules both singleton instructions and handles using the FU of the first instruction. If the handle belongs to an integer-memory mini-graph, a sliding-window scheduler ANDs the handle's **FUBMP** with its own current bitmap. If no bit in the result is set—*i.e.*, there are no downstream resource conflicts—the handle is scheduled, and its **FUBMP** is ORed into the current bitmap to make the reservations. If there is a conflict, handle issue is canceled and the slot used to attempt issue is lost. It is difficult to schedule multiple integer-memory handles in one cycle due to the need to cross-check the **FUBMP** of candidate handles against one another. Our experiments show that supporting the issue of a single heterogeneous handle per cycle is sufficient.

Partial mini-graphs on ALU pipelines. A sliding-window scheduler doesn't amplify execution bandwidth so any mini-graph processor will likely contain ALU pipelines. It is desirable to execute the contiguous integer portions of integer-memory mini-graphs on ALU pipelines. This is accomplished in our current scheme by proper definitions in the MGHT and MGST. The alternative definition of mini-graph 34 which schedules the last two operations on an ALU pipeline.

Mini-graph load scheduling. Integer-load mini-graphs—mini-graphs that contain loads—need not only be integrated with the register scheduler, but also with the load scheduler. There are two aspects to this integration: cache miss replays and memory disambiguation.

The MGT implicitly assumes a fixed latency for each instruction. What happens when a mini-graph load misses in the cache? There are two cases. Misses on terminal loads are handled like misses on singleton loads. No mini-graph instruction follows the load, so the scheduler holds (or replays) all waiting instructions (which may be younger handles) as usual. Misses on interior loads are more difficult. Since it is not possible to reschedule only the mini-graph subset that depends on the load, the entire mini-graph must be replayed. The result is a small performance penalty.

As in the case of branch prediction/update, a handle and its PC assume responsibility for memory disambiguation and load scheduling. Integer-load handles are scheduled according the same policy used to schedule singleton loads. These days, mechanisms like store sets [5]—which minimally synchronize loads and stores pair-wise—are popular. Like many other predictors, store sets is PC based and continues to work when loads and stores that are embedded in mini-graphs are identified by handle PCs rather than individual PCs. As on interior load misses, the entire enclosing mini-graph is (squashed and) replayed on a load mis-speculation.

5. Custom Mini-Graphs Using DISE

Mini-graphs capture common computational idioms. Some idioms are common to all programs, but the ability to program the MGT with application-specific mini-graphs is important. ***DISE (dynamic instruction stream editor)*** [8] effectively provides the programmable dynamic instruction set customization required to support application-specific mini-graphs. DISE is a facility for translating instructions into instruction sequences at decode time, according to programmable rewriting rules called productions. It is suitable for mini-graphs because mini-graph processors require handle-to-instruction-sequence translation but do not require further translation, *e.g.*, to FPGA directives.

DISE Primer. A DISE production is a **<pattern : replacement sequence>** specification pair. Pattern specifications can specify any combination of aspects of a single instruction: opcode, register name, or immediate. A replacement sequence is a sequence of instructions that is parameterized, *i.e.*, some fields in some instructions are “holes” to be filled in with field information from the matching instruction. The following toy DISE production **<add,–,–,–,– : T.INSN; andi T.RD,0xff,T.RD;>** injects after every **add** an instruction which clears all but the least-significant byte of the result. **T.INSN** and **T.RD** are template parameters. DISE examines every fetched instruction and replaces those that match active patterns with corresponding instruction sequence. Given the above production, the instruction **add r2,r4,r2** with the sequence **add r2,r4,r2; andi r2,0xff,r2**.

DISE has two usage modes. ***Transparent*** utilities operate on unmodified executables and redefine the semantics of naturally occurring instructions. Memory bounds checking is a example of a transparent utility; productions are defined for loads and stores. ***Aware*** utilities match and replace ***codewords***, quasi-instructions that are only meaningful to DISE and which have been planted into the executable by a DISE-aware compiler or binary rewriter. DISE codewords are recognized by their use of a reserved opcode; the codeword immediate serves as an index into the DISE on-chip translation structures. A DISE aware executable contains a special “.dise” section that defines the productions; the OS is responsible for loads this section into the DISE tables. Code decompression is an example of an aware utility.

DISE mini-graph productions. Mini-graph processing is an aware DISE utility and the format of a mini-graph handle matches that of a DISE codeword precisely. DISE provides a natural way for expressing the logical separation between a mini-graph’s register interface and its internal register dataflow. Interface registers are specified as parameters and are explicit in the codeword/handle. Mini-graph internal register dataflow is specified using DISE’s dedicated register set. This allows mini-graphs instantiated from different static handles not to interfere with local register definitions. The replacement sequences for our two mini-graphs are **<addl T.RS1,2,T.RD; cmplt T.RD,T.RS2,$d0; bne $d0, 0xa>** and **<ldq $d0, 16(T.RS2); srl $d0,14,$d0; and $d0,1,T.RD>**. **$d0** is a DISE register which denotes mini-graph interior dataflow.

A DISE mini-graph microarchitecture. Combining DISE and mini-graphs requires slight modifications to both. On the mini-graph side, we modify the MGT to act as a cache rather than a ROM, adding a small table, the ***mini-graph tag table (MGTT)*** to implement the tags. On the DISE side, we provide an option to keep codewords/handles un-expanded. Between the two, we add a small finite-state machine, the ***mini-graph preprocessor (MGPP)***, that scans DISE replacement sequences and compiles them to internal MGT format.

The DISE specification [8] implies that productions transform the instruction stream in-line, codewords are excised and replacement sequences spliced in their place. This design compartmentalizes DISE but also prevents the execution core from exploiting mini-graphs by feeding it a stream of singleton instructions. For mini-graph processing, we augment DISE with the option to forgo expansion and keep the codeword/handle inline. The decision to expand is based on finding the MGID in the MGTT. If the MGID is present—translation: this mini-graph is supported—the handle is not expanded. Otherwise, DISE expands the handle and the execution core processes each instruction individually. Keeping the expansion option preserves the correctness of DISE utilities whose productions do not meet mini-graph specifications. It also provides portability and compatibility at the intersection of mini-graph enabled executables and mini-graph processors. A processor can always expand a mini-graph it doesn't understand.

Each MGTT entry contains an MGID and two valid bits. The first valid bit simply indicates that a tag is not garbage and that the associated mini-graph has been pre-processed. The second valid bit indicates that the MGPP has "approved" the mini-graph and the handle should remain un-expanded. The MGTT is read at the DISE (decode) stage. On a miss, DISE expands the replacement sequence. One copy is sent to the execution core to avoid stalling the pipeline. A second copy goes to the MGPP for inspection/compilation.

6. Experimental Evaluation

We present a simulation-driven evaluation of mini-graph processing. We begin by studying mini-graphs functionally by examining the effects of constraints on mini-graph coverage. We follow with a performance evaluation, including sensitivity analysis.

Our simulators are constructed using the SimpleScalar Alpha AXP instruction and system call definition modules. The timing simulator models a 6-way superscalar, dynamically scheduled processor with a 15-stage pipeline, 128 entry reorder buffer, 64 entry load/store queue, and 50 entry issue queue. The execution engine uses a 164-entry, 5 read port, 4 write port, 2 cycle read physical register file. Each cycle, the scheduler may issue up to 6 operations with the following maximum composition: 4 integer, 2 floating-point, 2 load, and 1 store. Loads are scheduled using a store sets [5] predictor. Cache miss replays and memory ordering violation squashes are modeled faithfully.

We model a 12Kb hybrid branch direction predictor and a 2K-entry 4-way set-associative target buffer. The on-chip memory hierarchy includes 32KB, 2-way set-associative, 32B line 1-cycle access instruction and 2-cycle access data caches and a 2MB, 4-way set-associative, 128B line, 10-cycle access L2. Main memory has an access latency of 100 cycles and is accessed via a 16B bus that operates at one-quarter core frequency.

We use benchmarks from the SPEC2000, MediaBench [15], CommBench [25], and MiBench [10] suites. The benchmarks were compiled for the EV6 microarchitecture using the Digital OSF compiler with optimization level –O3. The SPECint programs were run on their training inputs at 2% periodic sampling with 10M instructions per sample; all other benchmarks were run unsampled on their largest available inputs. All benchmarks were run to completion. Results are shown for selected benchmarks along with means over all programs in each suite.

6.1. Coverage

Figure 5 shows coverage for integer (top) and integer-memory (middle) mini-graphs. Each bar group varies along the two MGT dimensions: total number of mini-graphs horizontally (32, 128, 512, 2K), and individual mini-graph size vertically (2,3,4,8).

With 512 integer mini-graphs, coverage averages 13%, 24%, 21%, and 19% for SPECint, MediaBench, CommBench, and MiBench, respectively. SPECint programs have both a lower ratio of ALU operations and smaller basic blocks. Integer-memory mini-graphs increase coverage by approximately 50%, to 21%, 33%, 31%, and 29%. Although a sliding window scheduler does not increase execution bandwidth, this increased coverage suggests that it should boost performance.

In practice, 60% of coverage is achieved using only 2 instruction mini-graphs. There is some advantage to allowing mini-graphs of size 3 and 4, but little benefit to allowing mini-graphs longer than that. Longer idioms that meet mini-graph criteria are simply not common; in SPECint the average basic block size is not much bigger than 4 instructions.

Most non-SPECint programs are statically so small that 128 MGT entries are sufficient to provide maximum coverage in all but a few cases (*ghostscript*, *rtr*). For SPECint, a similarly sized MGT will achieve maximum coverage for integer mini-graphs, but 512 entries are needed for integer-memory mini-graphs. 2K entries provide additional coverage for only a few programs (*gap*, *gcc*, *ghostscript*).

Intra-application input data robustness. Our off-line selection algorithm prioritizes mini-graph by coverage which is proportional to execution frequency. For maximum effectiveness, it relies on the robustness of basic block frequency profiles. We tested this robustness for SPECint and MiBench by selecting mini-graphs using basic block profiles from the *test* and *small* input data sets, respectively.

Our results (not shown) indicate that basic block frequency variance across input data sets reduces coverage relatively by an average of 15% (*e.g.*, from 20% to 17%). 70% of SPECint and 80% of MiBench programs

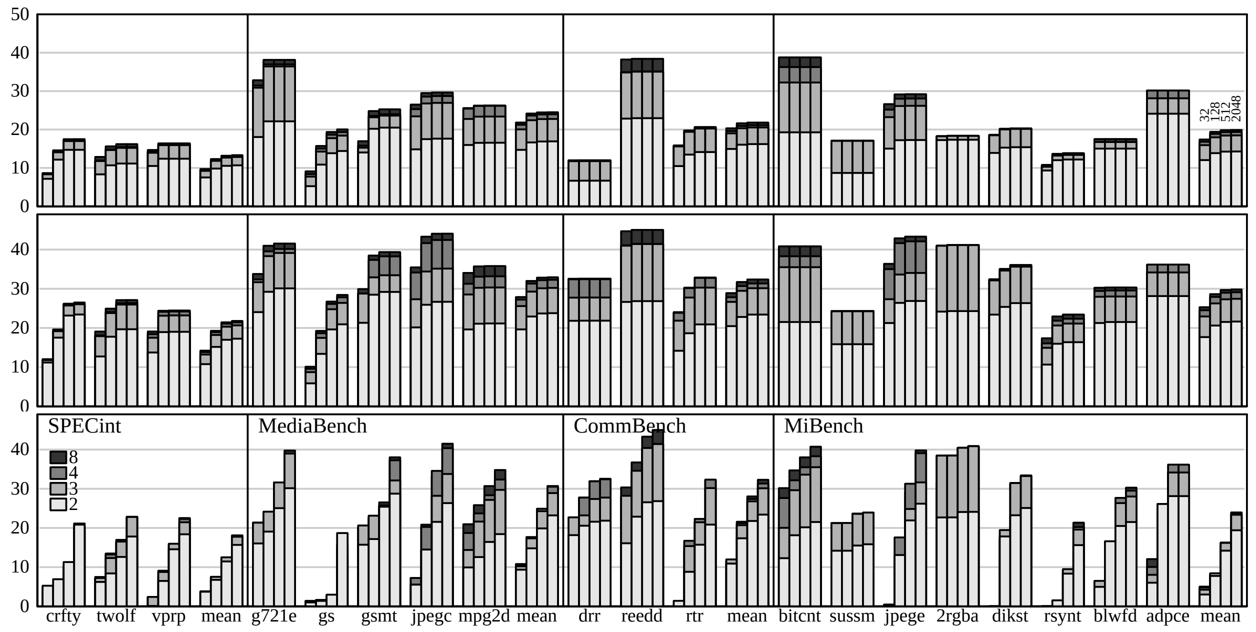


FIGURE 5. Coverage. (top) Application-specific integer and (middle) integer-memory, and (bottom) domain-specific integer-memory mini-graphs. Horizontal bars are 32,128, 512, 2K MGT entries. Stacks are mini-graphs of size 2,3,4, 8.

maintain coverages within 15% of those in Figure 5. However, several programs see their coverage drop to 0%. For these, different input files or flags simply trigger different portions of the static code. To avoid this pathology, mini-graph selection can and should incorporate profiles from multiple runs. In light of both the robustness of most programs and our proposed solution for less robust ones, we continue to use profiles generated from the same inputs used in testing.

Domain specific mini-graphs. The top two graphs in Figure 5 showed application-specific mini-graphs. The bottom graph shows coverage of domain specific integer-memory mini-graphs. Here, a 512 entry MGT holds the 512 most frequent mini-graphs across an entire benchmark suite. At 512 entries, each benchmark sacrifices some coverage for the benefit of the others in its suite. Intuitively, larger, 2K-entry MGTs are needed to achieve maximum per-application coverage. Interestingly, while two-instruction mini-graphs already dominate within an application, their relative contribution increases even further when they must cover multiple applications. Just as smaller idioms are more likely to be found in multiple static locations within a program, they are more likely to be found in different programs.

All subsequent experiments use an MGT that holds 512 application-specific mini-graphs with a maximum size of 4 instructions each.

6.2. Performance

Figure 6 shows the performance of two mini-graph processor configurations relative to our 6-wide baseline machine. In the first configuration, (light bars) integer mini-graphs execute on a similar pipeline in which two integer ALUs have been replaced with 4-stage ALU pipelines. In the second (dark bars), we further add a sliding-window scheduler capable of issuing one integer-memory mini-graph per cycle. Within each configuration there are two sub-configurations: the first (solid) uses simple ALU pipelines, the second (striped) uses pair-wise collapsing ALU pipelines. Baseline IPCs are printed below each benchmark.

We defer a discussion on latency-reducing ALU pipelines and focus on the resource amplifying configurations (solid portions of the bars). For integer mini-graphs and ALU pipelines, average (gmean) gains are 2% for SPECint, 10% for MediaBench, 6% for CommBench, and 7% for MiBench. For integer-memory mini-graphs and a sliding-window scheduler, those numbers are 2%, 12%, 3%, and 7%, respectively. There is a high degree of variance within each suite, with some programs (*reed.decode*, *mpeg2.decode*, *gsm.toast*) posting speedups of 20% and above while others show negligible gains or even losses (*crc*, *mcf*).

Isolating serialization effects. Performance losses manifest both for integer mini-graphs (*e.g.*, *mcf*, *drr*, *adpcm.encode*) and for integer memory mini-graphs (*e.g.*, *adpcm.rawc* whose performance improvement drops from 14% with integer mini-graphs to 11% with integer-memory mini-graphs). These losses are due to serialization—both internal and external—in the integer case and serialization and cache miss replays in the integer-memory case.

Figure 7 isolates the effects of these costs. The first

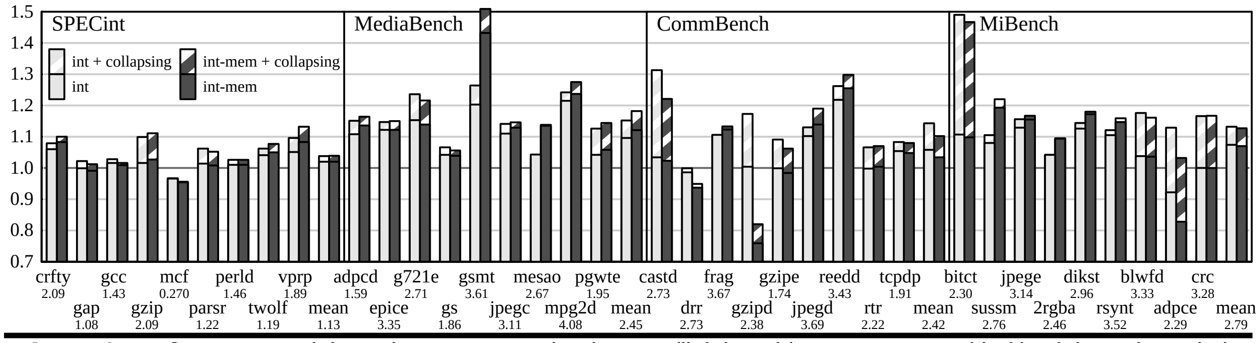


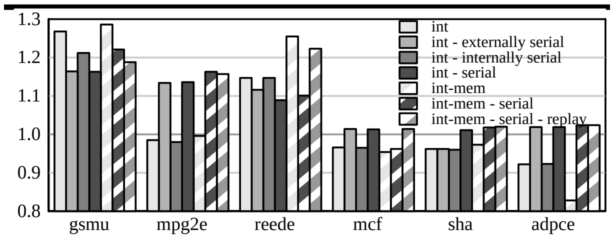
FIGURE 6. Performance. Mini-graph processor using integer (light) and integer-memory (dark) mini-graphs, relative to a baseline processor (baseline IPCs shown). Striped configurations use pair-wise collapsing ALU pipelines.

bar shows relative performance of integer mini-graphs for six programs; four of the six experience slowdowns. In the second bar, we eliminate external serialization by disallowing mini-graphs with external register inputs to any instruction other than the first, *i.e.*, whose first instruction may be spuriously delayed by inputs to subsequent instructions. For *gsm.untoast*, removing externally serial mini-graphs lowers performance gains from 27% to 16%. For *mpeg2.encode*, however, removing external serialization converts a 1% loss into a 13%gain.

In the third bar, we eliminate internal serialization by disallowing mini-graphs that have any internal parallelism, *i.e.*, are not serial dependence chains. This change has little effect in general because it can only possibly eliminate three and four instruction mini-graphs which are less common than two instruction mini-graphs. The only change we see is a slight performance loss in *gsm.untoast*. In the fourth bar, we disallow both externally and internally serial mini-graphs. *Sha* posts a speedup only when both types of serialization are eliminated.

The striped bars isolate effects for integer-memory mini-graphs. The first striped bar shows the performance of unconstrained mini-graphs. The second removes both internally and externally serial mini-graphs. Removing these two effects is enough to eliminate all performance degradation cases except for *mcf*, which suffers from load-miss mini-graph replays. In the final bar, we eliminate mini-graph replay effects by disallowing mini-graphs with loads in any position other than the last. Finally, *mcf*'s 4% performance loss returns to a 1% performance gain.

FIGURE 7. Serialization effects. Relative performance of mini-graphs with and without external serialization, internal serialization, and load-induced replays.



The non-uniform effects of each mini-graph selection sub-policy—allowing or disallowing internally serial, externally serial, and replay vulnerable mini-graphs—suggests that there is potential benefit to applying each policy selectively. Our results show that when the best combination of policies is applied on a per benchmark basis, average performance gains rise to 3%, 14%, 9% and 11%, respectively. Generally speaking, latency bound programs seem to prefer non-serializing mini-graphs while bandwidth bound programs can tolerate serialization and prefer increased coverage and bandwidth amplifications. Higher gains still may possibly be achieved if policies could be applied on a per mini-graph basis. We are currently investigating heuristics and profile measures for guiding the application of these and other policies to mini-graph selection. Our remaining experiments use unrestricted mini-graphs.

Latency reduction and resource amplification. Mini-graph processing is primarily a resource amplification technique. It is orthogonal to, but compatible with, dataflow-graph latency reduction. The striped portions of the bars in Figure 6 show experiments that add pair-wise collapsing to the 4-stage ALU pipelines of the corresponding mini-graph configurations. Two instruction integer mini-graphs execute in one cycle; three and four instruction graphs execute in two cycles. The addition of structured latency reduction boosts performance improvements to 4%,15%, 14%, and 13% for integer mini-graphs on ALU pipelines and 4%, 18%, 10%, and 13% for integer-memory mini-graphs on ALU pipelines and a sliding window scheduler. Generally speaking, latency reduction is less effective than bandwidth amplification, accounting for 30–50% of total performance improvement. However, it can provide a significant boost for latency bound programs like *bitcount* and *crc*. Again, the scope of latency reduction here is limited to dataflow graphs that meet mini-graph criteria.

Instruction cache effects. Like any static compres-

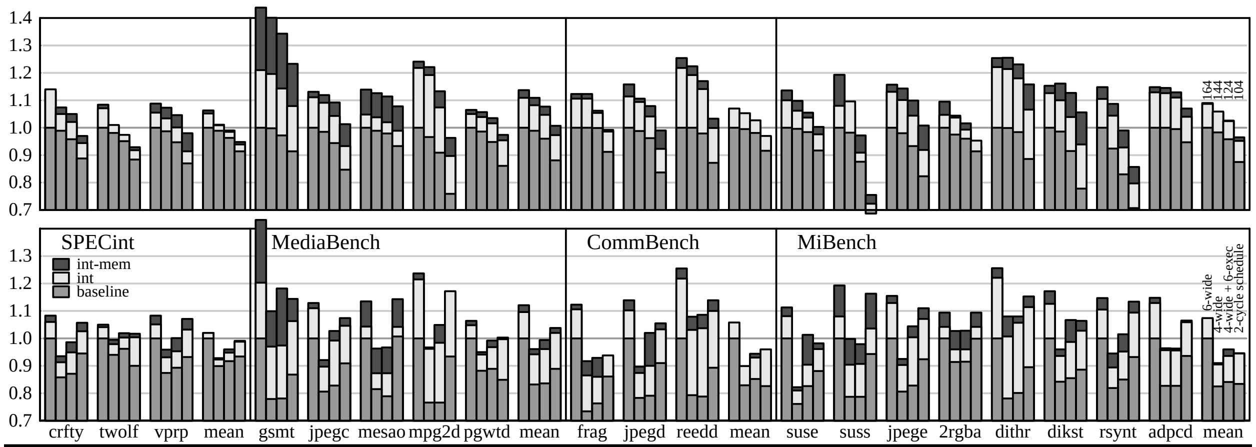


FIGURE 8. Resource amplification. (top) mini-graph performance with 144, 124, and 104 physical registers relative to a baseline with 164 registers. (bottom) mini-graph performance on a 4 wide processor, on a 4 wide processor with 6 execution units, and with a 2 cycle scheduler, relative to a baseline 6-wide processor with a 1 cycle scheduler.

sion technique, mini-graphs amplify instruction cache capacity. In order to isolate the effects of mini-graphs from those of ad hoc compression, none of our figures show the compression effect (we replace mini-graph interior instructions with nops). Because they have larger instruction footprints and working sets, SPECint programs are the only ones which experience a noticeable speedup from this effect. SPECint speedup triples to an average of 6%, while other suites gain on average less than 1% additional performance.

6.3. Resource Amplification as Simplification

The capacity and bandwidth amplification that mini-graphs provide can be used either to improve performance or to maintain performance at a lower complexity. We investigate the ability of mini-graph processing to compensate for reductions in physical register file size and bandwidth at all pipeline stages. We also measure its effectiveness at hiding scheduling loop latency. For this sensitivity analysis, we only use benchmarks for which mini-graphs provide a performance gain. This is done to ease data presentation. Means are still shown over all programs.

Capacity: physical register file size. The top graph in Figure 8 shows relative performance advantage of mini-graphs for processor configurations with reduced physical register files. Our baseline processor has 164 physical registers: 64 hold architected state and 100 hold in-flight state for a 128-entry reorder buffer (stores and branches are not allocated registers). We measure the effects of reducing the number of registers for in-flight instructions by 20%, 40% and 60% to 144, 124 and 104, respectively. For these reduced configurations, our baseline processor experiences average slowdowns of 1–2%, 2–4%, and 9–12%, depending on the benchmark suite. On average, mini-graphs can compensate—and often over-compensate—for a 40% reduction in physical registers. Intuitively, they cannot fully compensate for reductions that exceed coverage.

Although we do not show results, mini-graph processing can similarly deal with reductions in the number of scheduler (issue queue, reservation station) entries.

Bandwidth: all pipeline stages. Our baseline processor can fetch, rename, execute, and retire six instructions per cycle. The bottom graph in Figure 8 compares the effect of mini-graphs for that configuration (first bar) with their effect on two processors that can fetch, rename, schedule, and retire only 4 instructions per cycle. The first (second bar) can execute 4 instructions per cycle, including 1 load; the second (third bar) can execute 6 instructions per cycle, including 2 loads.

Relative to a 6-wide processor, a 4-wide processor represents performance degradations of 10%, 17%, 17%, and 18% for SPECint, MediaBench, CommBench, and MiBench, respectively. The addition of mini-graphs effectively restores much of this bandwidth; with mini-graphs, slowdowns are only 7%, 4%, 10%, and 9%, respectively. These remaining slowdowns are due to the fact that mini-graphs do not amplify execution bandwidth, specifically load execution bandwidth (ALU pipelines amplify integer execution bandwidth). When we restore the second load port, slowdowns are only 4%, 1%, 6%, and 4%.

Latency: scheduling loop. Our baseline processor models a “single-cycle” scheduler and can execute single-cycle operations and instructions that depend on them in consecutive cycles. Clock cycle concerns have forced some processors to pipeline the scheduler. The resulting wake-up/select loop disallows the issue of dependent instructions in back-to-back cycles and effectively increases the latency of all single-cycle operations to two cycles. Because mini-graph execution is “pre-scheduled” and does not use conventional wake-up/

select logic, it can help tolerate scheduling loop latency. This is the motivation for macro-op scheduling [14], a restricted micro-architectural precursor to mini-graph processing. Like macro-ops, mini-graphs hide scheduling latency in two ways: ***internally***, dependent instructions within the same mini-graph execute in consecutive cycles, and ***externally***, single cycle operations which impose a scheduling penalty on dependent instructions coalesce to multi-cycle operations which do not.

In the bottom graph of Figure 8, the right -most bar in each group shows a configuration with a two-cycle scheduler. In a conventional processor, two-cycle scheduling degrades performance by averages of 7–18% across the different benchmark suites. Mini-graphs compensate for 100% of this loss in MediaBench, 85% of it in SPECint, 80% in CommBench, and 70% in MiBench. These rates are roughly proportional to the mini-graph coverage with respect to single-cycle integer operations. Macro-op scheduling is reported to be more effective—it compensates for all by 0.5% of the performance loss on SPECint [14]—but this isn't surprising considering it specifically targets this problem. Macro-op scheduling's advantage here derives from its ability to exploit macro-ops that cross basic block boundaries.

7. Conclusions and Future Work

This work introduces mini-graphs, multi-instruction dataflow graphs with an instruction-like interface: two inputs, one output, at most one memory reference, and at most one control transfer. We detail a microarchitecture that processes entire mini-graphs via a single quasi-instruction handle. Mini-graphs reduce bandwidth consumption at every pipeline stage and storage demands on resources like the scheduler and register file.

Our results show that mini-graphs can cover 10–50% of the dynamic instruction stream. Relative to an aggressive baseline, they achieve average performance gains of 2%, 12%, 6% and 7% on SPECint, MediaBench, CommBench and MiBench respectively, with peak gains exceeding 30% and 40%. Alternatively, they can also effectively compensate for 40% reductions in register file and scheduler sizes, 33% reductions in pipeline bandwidth, or a pipelined scheduler. Our analysis shows that there is room for further improvement.

Future work will focus on selection heuristics and on the energy properties of mini-graph processors.

Acknowledgments

We thank the anonymous reviewers for their suggestions. E Lewis, Milo Martin, and Ronny Ronen gave comments that helped improve this manuscript. This work was supported by NSF CAREER award CCR-0238203 (Roth), NSF award CCR-0311199, an NSF Fellowship (Bracy), and a generous gift from Intel.

References

[1] K. Atasu, L. Pozzi, and P. Ienne. "Automatic Application Specific Instruction Set Extensions Under Microarchitectural Constraints." In *DAC-40*, Jun. 2003.

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

[3] E. Borch, E. Tune, S. Manne, and J. Emer. "Loose Loops Sink Chips." In *HPCA-8*, Jan. 2002.

[4] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. "Instruction Generation and Regularity Extraction for Reconfigurable Processors." In *CASES-02*, Oct. 2002.

[5] G. Chrysos and J. Emer. "Memory Dependence Prediction using Store Sets." In *ISCA-25*, Jun. 1998.

[6] 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 *MICRO-37*, Dec. 2004.

[7] N. Clark, H. Zhong, and S. Mahlke. "Processor Acceleration through Automated Instruction Set Customization." In *MICRO-36*, Dec. 2003.

[8] M. Corliss, E. Lewis, and A. Roth. "DISE: A Programmable Macro Engine for Customizing Applications." In *ISCA-30*, Jun. 2003.

[9] D. Goodwin and D. Petkov. "Automatic Generation of Application Specific Processors." In *CASES-03*, Oct. 2003.

[10] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. "MiBench: A Free, Commercially Representative Embedded Benchmark Suite." In *WWC-4*, Dec. 2001.

[11] J. Hauser and J. Wawrzynek. "Garp: A MIPS Processor with a Reconfigurable Coprocessor." In *FCCM-97*, Apr. 1997.

[12] S. Hu and J. Smith. "Using Dynamic Binary Translation to Fuse Dependent Instructions." In *CGO-2*, Mar. 2004.

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

[14] I. Kim and M. Lipasti. "Macro-op Scheduling: Relaxing Scheduling Loop Constraints." In *MICRO-36*, Dec. 2003.

[15] C. Lee, M. Potkojnak, and W. Mangione-Smith. "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems." In *MICRO-30*, Dec. 1997.

[16] N. Malik, R. Eickemeyer, and S. Vassiliadis. "Interlock Collapsing ALU for Increased Instruction-Level Parallelism." In *MICRO-25*, Dec. 1992.

[17] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler. "A Design Space Evaluation of Grid Processor Architectures." In *MICRO-34*, Dec. 2001.

[18] S. Patel and S. Lumetta. "rePLay: a Hardware Framework for Dynamic Optimization." *IEEE Transactions of Computers*, 50(6), Jun. 2001.

[19] A. Peymandoust, L. Pozzi, P. Ienne, and G. D. Micheli. "Automatic Instruction Set Extension and Utilization for Embedded Processors." In *ASAP-14*, Jun. 2003.

[20] J. Phillips and S. Vassiliadis. "High-Performance 3-1 Interlock Collapsing ALUs." *IEEE Transactions on Computers*, 1994.

[21] R. Razdan and M. Smith. "A High-Performance Microarchitecture with Hardware Programmable Function Units." In *MICRO-27*, Dec. 1994.

[22] Y. Sazeides, S. Vassiliadis, and J. Smith. "The Performance Potential of Data Dependence Speculation and Collapsing." In *MICRO-29*, Dec. 1996.

[23] J. Stark, M. Brown, and Y. Patt. "On Pipelining Dynamic Instruction Scheduling Logic." In *MICRO-33*, Dec. 2000.

[24] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. "WaveScalar." In *MICRO-36*, Dec. 2003.

[25] 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.

[26] Z. Ye, A. Moshovos, S. Hauck, and P. Banerjee. "CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit." In *ISCA-27*, Jun. 2000.

[27] S. Yehia and O. Temam. "From Sequences of Dependent Instructions to Functions: A Complexity-Effective Approach for Improving Performance without ILP or Speculation." In *ISCA-31*, Jun. 2004.