Exploiting Large Ineffectual Instruction Sequences

Eric Rotenberg

Abstract

A processor executes the full dynamic instruction stream in order to compute the final output of a program, yet we observe equivalent, smaller instruction streams that produce the same correct output. Based on this observation, we attempt to identify large, dynamically-contiguous regions of instructions that are ineffectual as a whole: they either contain no writes, writes that are never referenced, or writes that do not modify the value of a location. The architectural implication is that instruction fetch/execution can quickly bypass predicted-ineffectual regions, while another thread of control verifies that the implied branch predictions in the region are correct and that the region is truly ineffectual.

1. Introduction

A general purpose program is a specification to the processor executing that program: a "contract" of the work to be performed and the output that must ultimately be produced. But the specification makes no requirement of *how* the processor should reach the final, correct state, and this allowance has resulted in many microarchitecture innovations (e.g., out-of-order execution, multiple instruction issue, branch and value prediction/speculation) that transform an apparently slow, sequential program into a faster, parallel one.

This paper suggests the possibility of identifying a smaller fraction of dynamic instructions that, when executed alone, produce the same overall effect as executing all of the specified dynamic instructions. The concept (Figure 1) is only an exercise at this point and is based on observing the evolution of a running program ideally. Nevertheless, the exercise is revealing: in

some cases, equivalent instruction streams comprising as little as 20% of the full run produce the correct final program state. This result is based on the following observations.

- Some instructions write a value to a register or memory location and the value is overwritten before ever being used, or even if not overwritten, is simply never referenced. Such instructions, and the computation chains leading up to these instructions, have no effect on final program state.
- Some instructions write the same value into a register or memory location as already exists at that location. Such instructions, and the computation chains leading up to them, have no observable effect on final program output because their writes were not truly modifications.
- The effects of branches (and the computation chains feeding the branches) are bypassed when their outcomes are predicted in advance. The non-trivial case, and the one we are primarily interested in, is a long sequence of correctly-predicted branches that either produces no writes, or produces only ineffectual writes.

The term “ineffectual” is used to describe instructions that fall into any of the above categories. The term “ineffectual” does *not* mean the computation is unnecessary or avoidable, as will be shown later. Our study of ineffectual instruction sequences progresses in four basic steps; performance and architecture implications become clearer with each step.

1. Individual dynamic instructions are separated into two categories, *effectual instructions* and *ineffectual instructions*. The purpose of this initial step is to establish techniques for ideally (but not optimally) identifying ineffectual instructions, and propagating ineffectual status backward through their dependence chains. The techniques are validated by stripping the full dynamic instruction stream of all ineffectual computation, and verifying the final output of the

much-reduced dynamic program. The number and type of ineffectual instructions are measured in this phase.

2. Next, runs of ineffectual instructions are identified. Dynamically-contiguous, **ineffectual regions (IR)** are relevant because they suggest architectures that learn, predict, and exploit entire IRs as a single unit. We characterize key properties of IRs: IR lengths; number of unique IRs and their repetition, or dynamic frequency; and the *ineffectual rate*, or fraction of occurrences for which a particular dynamic instruction sequence is ineffectual. The degree of repetition and the ineffectual rate will no doubt impact the predictability of IRs.

3. We suggest ways of exploiting IRs and propose new architectures based on the concept. The underlying idea is to speculatively skip past IRs, i.e., quickly re-route instruction fetching to the point just after the IR. The IR (if predicted correctly) does not modify state in any relevant way, so execution can proceed beyond the IR using the register/memory state immediately prior the IR. In other words, IRs provide a natural source of parallel threads because they create no dependences with subsequent computation. Of course, the IR may still have to be executed to confirm that 1) the implicitly-predicted control flow through the IR is correct and 2) the region is indeed ineffectual.

   Our proposed approach leverages the trend of simultaneously running multiple independent threads on the same chip [1,2]. We consider both a single-chip symmetric multiprocessor (SMP) [2] and a simultaneous multithreaded processor (SMT) [1], in which a primary thread freely skips over predicted IRs and a second redundant thread executes all instructions and validates the leading thread [3]. The rationale of each of these architectures is explained, e.g., a 2-way SMP is an attractive alternative to doubling superscalar complexity for speeding up a single program.

4. An IR-based architecture requires several new mechanisms. Key mechanisms are briefly discussed in the context of the SMP/SMT architectures, although our research is in its early stages.

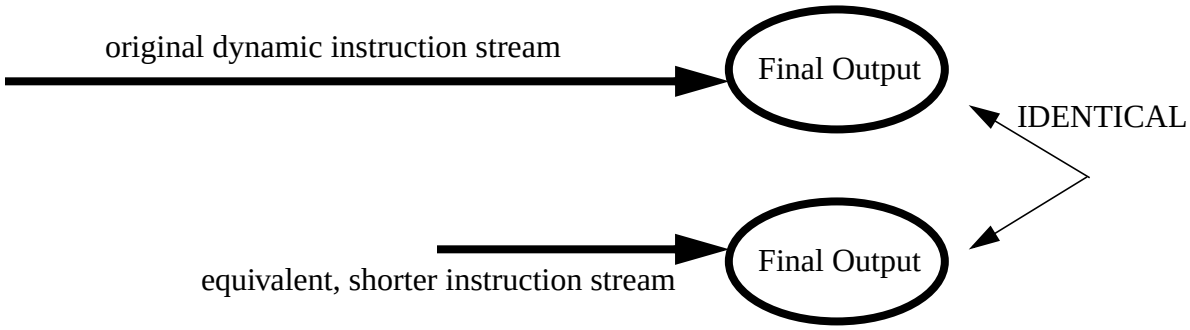

FIGURE 1. Exercise: finding a much-reduced instruction stream that produces the same final output.

2. Related work

Researchers have demonstrated a tremendous amount of redundancy, repetition, and predictability in general purpose programs. Lipasti, Wilkerson, and Shen [4] and Sodani and Sohi [6,7] showed that instructions tend to consume and produce the same values repeatedly. This property can be exploited to collapse entire chains of computation into a single cycle of execution, either speculatively based on value prediction [5] or non-speculatively via instruction reuse [6].

Huang and Lilja extended instruction reuse to basic blocks [8] and Gonzales, Tubella, and Molina proposed trace-level reuse [9], i.e., reusing previously-computed live-outs of an *arbitrarily large* dynamic sequence of instructions given repeated live-in operands to the trace. In addition to executing a long chain of instructions in a single or a few cycles, trace-level reuse is especially interesting because it potentially eliminates fetching the trace altogether. In their modeling of program predictability, Sazeides and Smith [10] suggest speculatively bypassing large dynamic sequences of instructions.

The microarchitecture trend of quickly bypassing large regions of work is the primary motivation behind ineffectual regions. Ineffectual regions are a special case, however, where state does not need to be updated at all. Furthermore, although we propose quickly bypassing ineffectual regions, they are still executed (our multithreaded architecture for exploiting ineffectual regions is based on AR-SMT [3]). For both these reasons (no trace outputs and computation instead of reuse), we avoid the state explosion that trace-level reuse is potentially prone to.

3. Analysis of ineffectual instructions and regions

3.1 Methodology

Ineffectual instructions are identified ideally by executing the program and using the resulting trace to construct a dynamic dataflow graph. Adding a new instruction to the graph requires identifying the producers of any source operands. This is facilitated by a register table and an unbounded memory table that indicate the most recent producer of a given register or memory location, respectively. Additional state is maintained within each register/memory table entry, and within each node (i.e. instruction) in the dataflow graph, to detect and propagate ineffectual status.

When a new instruction is added to the graph, it may cause the detection of an ineffectual instruction. The graph generator checks three conditions to detect an ineffectual instruction.

1. If the new instruction writes into a register/memory location, the value written by the previous producer to that location is “killed” (the old value can no longer be referenced by any new instructions). If the old value had never been referenced, that value is marked “ineffectual” in the previous producer’s node. If this is the only value produced by the node, or if all other values produced by the node are also marked “ineffectual”, then the previous producer instruction can be considered ineffectual.

2. The value produced by a new instruction is compared to the value written by the previous producer to the same location. If the values match, then the value written by the *new instruction* is marked “ineffectual” in the *new instruction’s* node. If this is the only value produced by the node, or if all other values produced by the node are also marked “ineffectual”, then the new instruction can be considered ineffectual.

3. All correctly predicted branch instructions are considered ineffectual. (The branch predictor is described in Section 3.2.)

The first and second conditions may be true simultaneously, in which case the second condition arbitrarily takes precedence. The third condition is necessary for 1) detecting ineffectual branch-related computation, as defined in the introduction, and 2) identifying branch-predictable IRs containing possibly many basic blocks. Regarding the latter, if predictable branches were not considered ineffectual, than IRs would be unnecessarily short in length. Also, ineffectual branches are only interesting within the context of larger IRs, and our results should be interpreted with this understanding in mind.

So far we have described detecting the *sources* of ineffectual behavior. When an instruction is marked “ineffectual”, the dataflow graph generator attempts to propagate ineffectual status backward along dependence arcs. Each source operand of the ineffectual instruction is processed as follows. The producer of the source operand is identified, and the corresponding value in the producer node is marked “ineffectual” if 1) the value has been killed, i.e., all uses of the value are fully known, and 2) all other uses have indicated similar “ineffectual” status. The value may be killed at a later time, in which case the same check is invoked at that time. When all values created by the producer instruction are marked “ineffectual”, the producer instruction itself is considered ineffectual, and the process continues recursively.

3.2 Simulation environment

An important restriction of the study is that the dataflow graph cannot grow arbitrarily large. A size of 64K (65536) nodes is maintained at all times. When a new instruction is incorporated into the graph, the node corresponding to the oldest instruction is removed, at which time its ineffectual status is checked. An unknown fraction of ineffectual instructions will be considered effectual as a result of the limited graphs. In Section 3.3, we measure the number of ineffectual instructions detectable with graphs smaller than 64K.

A 64K-entry *gshare* predictor [11] is used to predict conditional branch outcomes. A 64K-entry target predictor, using the same *gshare* index, is used to predict the targets of jump indirect and call indirect instructions [12] (direct branch targets require no prediction). An unbounded return address stack [13] is used to predict subroutine return instructions.

The Simplescalar toolset [14] is used to generate instruction traces. Binaries were compiled with -O3 level optimization. The Simplescalar compiler is gcc-based and the ISA is MIPS-based; as a result, instruction traces inherit any inefficiencies of the gnu compiler and MIPS ISA. In the future, we plan on evaluating the impact of the compiler, and we hope to better understand how much ineffectual behavior is due to the programmer/algorithm, the language, the ISA, and the compiler.

A total of six benchmarks are used -- five integer SPEC95 benchmarks and the *postgres* database backend -- as shown in Table 1. *Postgres* version 6.4.2 [15], ported to the Simplescalar toolset [16], runs TPC-R query #1 on a scaled-down version of the TPC-R database [17].

We verified correctness of ineffectual instruction analysis using the process depicted in Figure 1. The full run of the program is stripped of all ineffectual instructions and the resulting shortened trace -- program counter values only -- is fed into the *sim-fast* functional simulator of

the Simplescalar toolset. The program counters merely override the normal next-PC computation of *sim-fast*. For all benchmarks, 1) the reduced program completes without prematurely faulting, core-dumping, etc., and 2) the final output matches the final output of the full run.

TABLE 1. Benchmarks.

benchmark	input	instruction count	branch misp. rate
gcc	-O3 genrecog.i -o genrecog.s	117 million	7.77%
go	9 9	133 million	15.53%
jpeg	vigo.ppm	166 million	6.67%
m88ksim	-c < ctl.in	119 million	1.55%
perl	scrabbl.pl < scrabbl.in	108 million	2.71%
postgres	1.sql	423 million	1.42%

3.3 Results for ineffectual instructions

In this section, we measure the number of individual instructions that are effectual and ineffectual. Ineffectual instructions are further broken down into those that are the source of ineffectual behavior and those that are ineffectual because their values are used only by other ineffectual instructions. A breakdown of all dynamic instructions is graphed in Figure 2 using the following notation.

- `effectual`: The fraction of dynamic instructions that are effectual. These instructions, when run alone, produce the same effect as running the entire program.
- `BR`: The first of three *sources* of ineffectual behavior -- correctly predicted branches.
- `WW`: The second of three sources of ineffectual behavior -- a write followed by a write to the same location, with no intervening reference.
- `SV`: The third source of ineffectual behavior -- writing the same value to a location.

- `P:{BR | WW | SV}`: There are seven categories for instructions that inherit, via back-propagation, ineffectual status from subsequent ineffectual instructions. The `BR`, `WW`, and `SV` qualifiers indicate *all* sources of ineffectual behavior that an instruction has inherited.
- `other`: Most sources of ineffectual behavior fall into only one of the `BR`, `WW`, and `SV` categories. It is unusual, but possible, for a source to be ineffectual for more than one reason. The `other` category includes 1) ineffectual store instructions that store multiple bytes, some of which are attributed to `WW` and others to `SV`, and 2) call instructions that link the return address through a register, in which case both `BR` and `SV` can be simultaneously active. This category is infrequent and is only included for completeness.

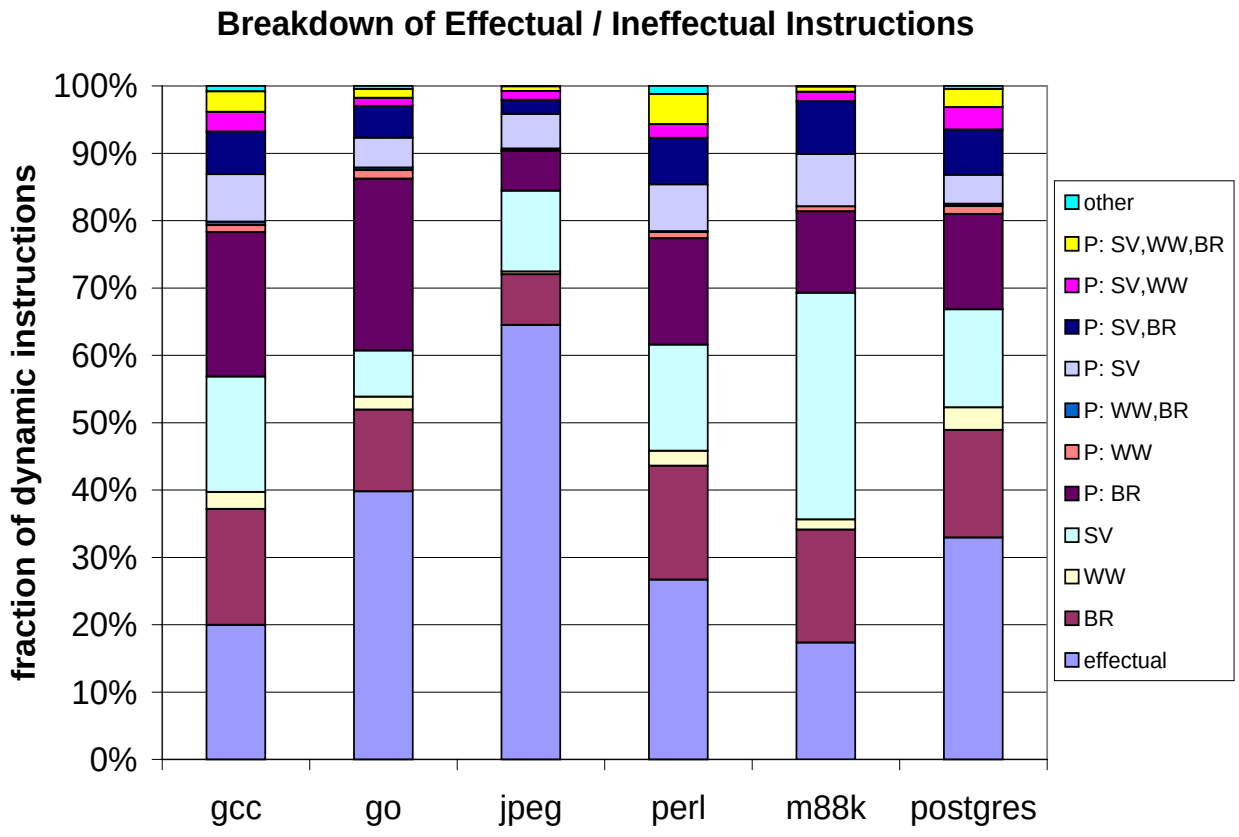

FIGURE 2. Breakdown of effectual and ineffectual instructions.

For *gcc* and *m88k*, fewer than 20% of the dynamic instructions are effectual; *perl* and *postgres* are about 30% effectual; *go* and *jpeg* are 40% and 65% effectual, respectively. This confirms our hypothesis that there exist significantly smaller, equivalent dynamic instruction streams that produce the correct, final program state.

Of course, we must interpret this result carefully. Branch instructions account for approximately 1 of every 5 instructions in integer benchmarks. Consequently, correctly-predicted branches (`BR`) directly account for 20% (*go*) to 24% (*postgres*) of ineffectual instructions, and indirectly account for 15% (*m88ksim*) to 40% (*go*) of ineffectual instructions as a result of computation feeding the branches (`P:BR`).

Unreferenced writes, `WW`, appear to be a small source of ineffectual behavior, however, this is most likely an artifact of `SV` having precedence when both occur simultaneously (other results, not presented here, indicate `WW` is a large source when given precedence). Together, `WW` and `SV` directly account for 15% (*go*) to 43% (*m88ksim*) of ineffectual instructions, and indirectly account for about 15% additional ineffectual instructions (`P:WW`, `P:SV`, and `P:WW,SV`).

As described in Section 3.2, the dataflow graph used to identify ineffectual instructions contains 64K nodes. Figure 3 shows that 80% (*perl*) to 95% (*m88ksim*) of ineffectual instructions are identified within a distance of 256 nodes, i.e., a significantly smaller dataflow graph can achieve similar results.

The full implications of ineffectual behavior are not clear until we consider groups of ineffectual instructions, or IRs, which we treat in the next section. An individual, correctly-predicted branch, along with a few isolated instructions in the branch's program slice, do not suggest an exploitable program property. Rather, we are interested in cases of large, repetitive dynamic code sequences that execute and produce no observable output; *predicting the sequence, in fact, implies there will be no changes to persistent program state*. An example is shown in Figure 4(left), in

which a long linked-list search executes only to verify the original assumption: the element is not in the list (`found = false;`). If this outcome is repetitive in a predictable way, the entire loop can be speculatively bypassed because it is ineffectual.

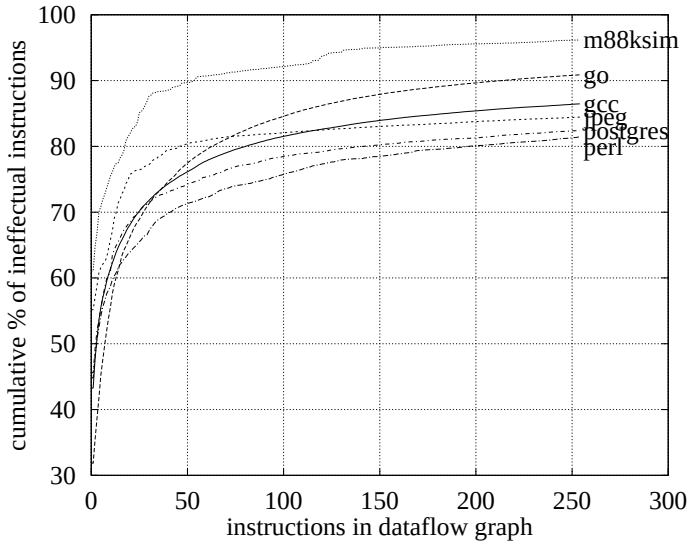

FIGURE 3. Size of dataflow graph and its impact on identification of ineffectual instructions.

```
found = false;
ptr = head;
while (ptr) {
    if (ptr->key == key) {
          ···never entered
    }
    else
      ptr = ptr->next;
}
```

```
s = connect_bamboo(...);
if (s != NO_SQUARE) {
  ... never entered
}
```

FIGURE 4. Linked-list example of a large ineffectual region.

The benchmark *go*, incidentally, exhibits the example because it performs many list searches (Figure 4(right)). For example, we observed many occurrences of an IR, 92 instructions in length, corresponding to an entire dynamic instance of the `connect_bamboo()` function; the IR corresponds to a search that fails and no further work is performed.

3.4 Results for ineffectual regions

In this section, we identify dynamically-contiguous groups of ineffectual instructions. We are particularly interested in long IRs (length-20 and longer), so we first measure their combined contribution to ineffectual instruction count. Next, we identify *unique* IRs and measure their individual contributions to ineffectual instruction count.

The graphs in Figure show the cumulative fraction of ineffectual instructions due to length-1 IRs, length-2 IRs, etc. For *gcc* and *m88ksim*, length-20+ IRs contribute about 40% of all ineffectual instructions; this is followed by *go* with 35%, *perl* with 30%, and *jpeg* and *postgres* with 20%.

IRs, like traces [19,20], are uniquely identified by a start PC and a sequence of embedded branch outcomes. In the following analysis, we assume IRs are terminated at indirect branches, similar to conventional trace selection [19,20] (the previous analysis assumes no such constraint, and there is a noticeable discrepancy between Figures 5 and 6 for *m88ksim*). We identify all unique IRs containing 20 or more instructions and collect three pieces of information for each IR: 1) length, 2) number of times the dynamic sequence is encountered, and 3) the number of times the dynamic sequence is ineffectual. The product of 3) and 1) is the IR's individual *contribution* to ineffectual instruction count. The ratio of 3) divided by 2) is the *ineffectual rate*, i.e., the fraction of occurrences that the IR was ineffectual. In Figure 6, IRs are sorted by decreasing *contribution* or decreasing *ineffectual rate* and, progressing down the sorted list, we accumulate the fraction of ineffectual instructions (y-axis) and the fraction of unique IRs (x-axis).

Sorting by decreasing *contribution* tries to maximize the number of ineffectual instructions with the minimal number of unique IRs. For *gcc*, only 5% of length-20+ IRs are needed to provide 30% of *gcc*'s ineffectual instruction count; all length-20+ IRs taken together can only pro-

vide 38%. The same trend applies for the other benchmarks (except *perl*) -- 5% of length-20+ IRs provide a majority of the benefit of all length-20+ IRs.

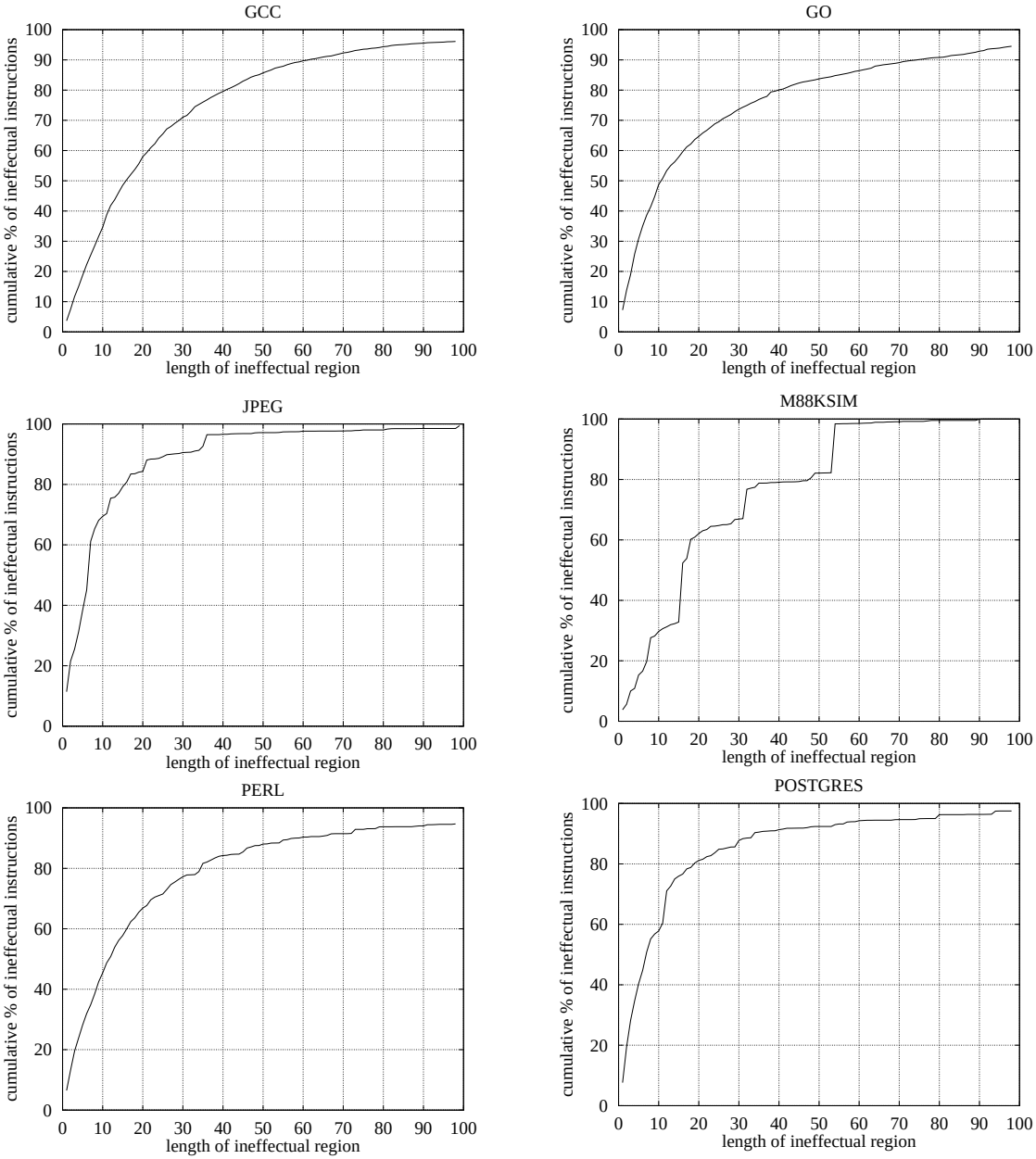

FIGURE 5. IR length and contribution to ineffectual instruction count.

Sorting by decreasing *ineffectual rate* places more value with IRs that are consistently ineffectual. An IR that is ineffectual 100% of the time is likely to be more predictable than an IR that

is ineffectual 50% of the time (although the latter may be just as predictable). Therefore, the second curve in each graph of Figure 6 may be more representative of expected “returns” from individual IRs. We can see that IRs with high *ineffectual rates* do not necessarily contribute many ineffectual instructions and, conversely, IRs that make large contributions may not be consistently/predictably ineffectual.

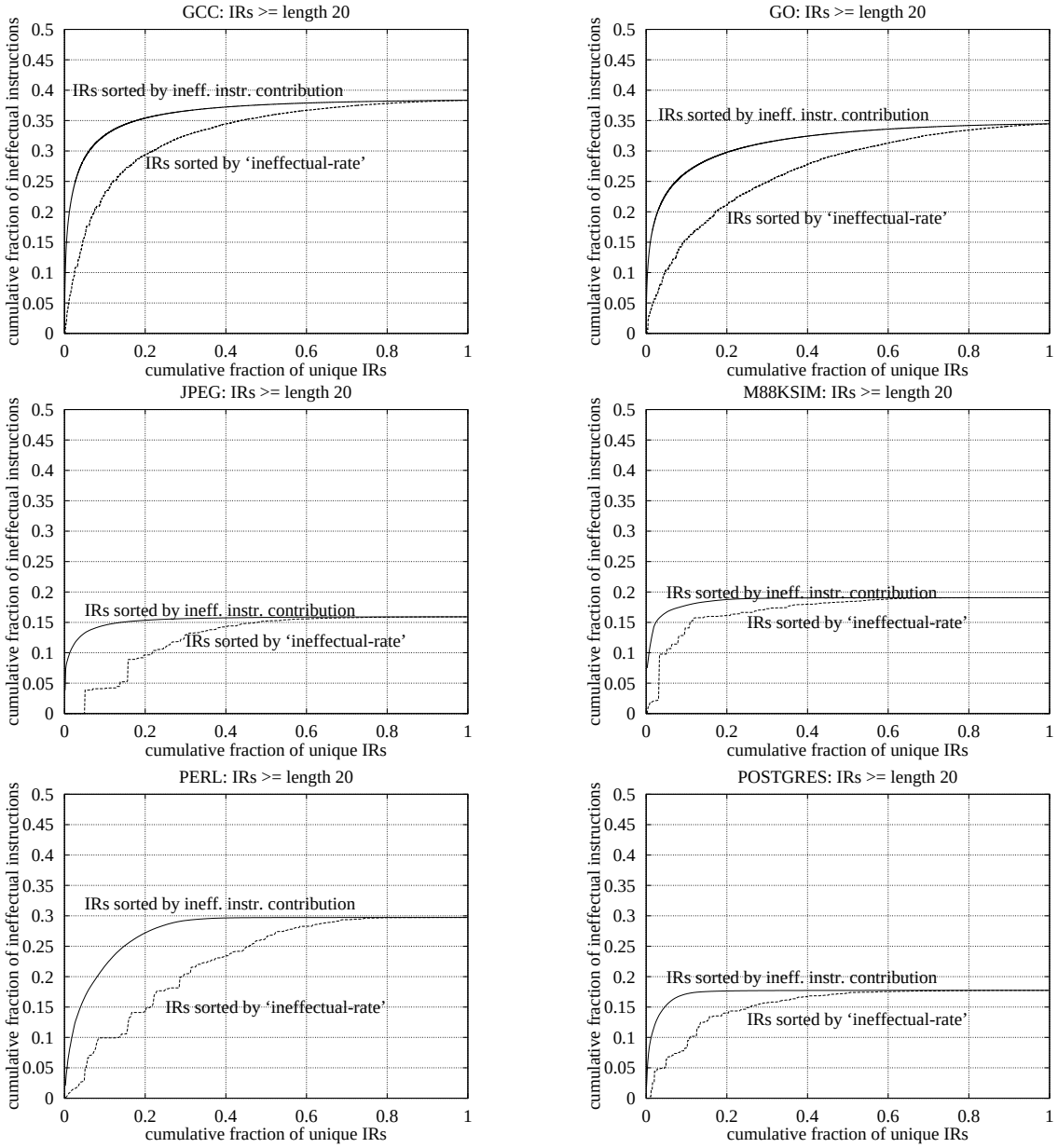

FIGURE 6. Unique IRs and contribution to ineffectual instruction count.

4. Architectural implications

In this section, we suggest an unconventional approach for speeding up programs. The approach leverages ineffectual regions and multithreading/multiprocessing execution models.

Single-chip, symmetric multiprocessors (SMP) [2] and simultaneous multithreading processors (SMT) [1] exploit both thread-level and instruction-level parallelism, effectively utilizing processor resources. And we feel SMP/SMT architectures provide a unique opportunity to explore new paradigms. The opportunity stems from 1) the close proximity, in both space and time, of multiple threads, 2) the separation of state that enables autonomy among threads, and 3) the flexibility to dynamically choose from among multiple execution modes using the same hardware, depending on the application and need at hand (e.g., single-thread performance, high utilization, reliability [3]).

4.1 Cooperating, partially-redundant threads

Exploiting ineffectual regions implies skipping over the regions. Unfortunately, we cannot guarantee ahead of time that a dynamic sequence of instructions is ineffectual, for several reasons. First, there are implicit control flow predictions through the region. Second, there are implicit data predictions in the region, for example, if ineffectual behavior stems from overwriting the same value in a location. Third, *future* control flow and data flow ultimately determines ineffectual behavior.

Therefore, skipping over IRs is speculative in nature and validation is required. We propose running two partially-redundant threads on either an SMP or SMT processor. At any given time, one of the threads is considered the primary or *active* instruction stream (A-stream). The A-stream fetches and executes instructions in a conventional manner, but it also predicts future

IRs based on the past history of IRs. When an IR is predicted, the A-stream transparently deviates from conventional fetch/execution by skipping past the ineffectual region.

The second thread is a redundant instruction stream (R-stream) that runs “behind” the A-stream. It maintains its own copy of program state. Because it executes behind the A-stream, the R-stream has the benefit of perfect control and data flow “predictions” obtained from the A-stream and, consequently, executes with peak parallelism. Until the A-stream predicts an IR, the R-stream functions merely to keep in step with the A-stream and maintain a correct copy of program state.

When the A-stream skips past an IR, the R-stream begins executing the IR in a conventional manner (it has no *data flow* predictions from the A-stream because the A-stream skipped over these instructions). Meanwhile, the A-stream is free to slip far ahead of the R-stream. When the R-stream completes execution of the IR, it once again has perfect predictions available from the A-stream. Therefore, the R-stream has the opportunity to catch up to the A-stream by executing instructions with peak parallelism.

Figure 7 shows a high-level view of the cooperating A-stream and R-stream. The thick lines indicate the dynamic instruction streams. Time and execution progress from left to right. Parts of the R-stream that execute with peak parallelism -- due to values and branch outcomes passed from the A-stream -- are shown with rotated lines (indicating many instructions executed in parallel). In the center of the figure, the A-stream encounters a predicted IR and skips past it. The R-stream executes it in less-than-peak-parallel fashion, but is able to catch up with the A-stream after executing the IR.

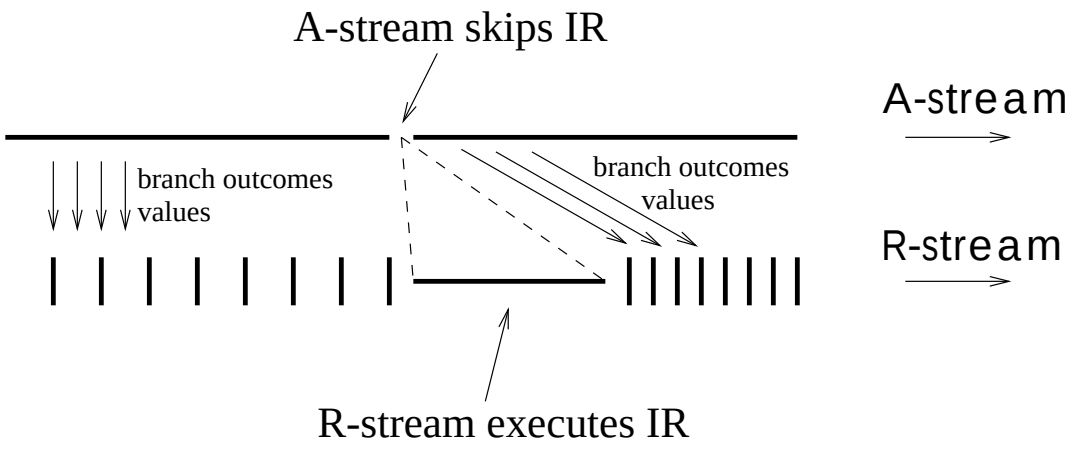

FIGURE 7. Cooperating threads for exploiting ineffectual regions.

Because the R-stream executes all instructions, it is able to 1) detect IR mispredictions in the A-stream and 2) continuously update the state of the IR prediction mechanism. In fact, the two functions are performed by the same hardware since updates and validation both require searching for ineffectual regions.

Note that IR-mispredictions can be detected at three different points in time. First, if the IR is incorrect due to mispredicted control flow, this will be detected during execution of the IR by the R-stream. When the A-stream predicts an IR, it provides control flow information about the predicted IR to the R-stream (start PC and branch outcomes through the region, similar to trace prediction [18]). This enables the R-stream to validate the predicted IR's control flow. Second, if the predicted IR is not actually ineffectual, and instructions immediately after the skipped region depend on values produced in the region, the R-stream will detect branch/value mispredictions coming from the A-stream. Third, if the predicted IR is effectual but the effects are not detectable for some time (i.e., the A-stream still appears uncorrupted), the IR-detection hardware will eventually detect that the region is effectual.

A-stream state is corrupted in the event of an IR-misprediction. And although the R-stream state is correct, there is no obvious method for repairing A-stream state using R-stream state because the differences are unknown. The solution is to maintain a history of writes in the A-stream. When a write occurs to a location, the old value is saved in the history buffer in the event we need to recover the old value. Subsequent writes to the same location need not save previous values. After detecting a misprediction, A-stream state is selectively repaired by reading old values from the history buffer. (Note that a single history buffer can easily support any number of outstanding, unresolved IR predictions.)

Lastly, when an IR misprediction is detected in the A-stream, the roles of the A-stream and R-stream swap. The A-stream must back up just prior to the IR, whereas the R-stream may have completed executing the IR and instructions beyond it. Therefore, the (old) R-stream is actually ahead of the (old) A-stream.

We now summarize the key, new components of the architecture:

- A mechanism for detecting and selecting IRs.
- An IR predictor.
- A buffer for communicating control and data flow predictions from the A-stream to the R-stream. The R-stream uses these predictions to execute with peak parallelism and it validates the predictions with computed results (catching many IR mispredictions before the IR detection mechanism).
- A write history buffer for repairing corrupted A-stream state in the event of an IR-misprediction.

4.2 Performance of cooperating threads with an ideal IR implementation

In this section, we demonstrate the ability to obtain higher performance using the cooperating thread concept. The intent is to validate the execution model's performance claims.

4.2.1 Simulator

We use an ideal IR implementation to test the execution model. IRs are identified using the method of Section 3. *Only length-20 or longer IRs are exploited*. The *gshare* predictor of Section 3 is used to predict control flow; IR prediction itself is perfect, i.e., given a branch-predictable dynamic sequence of instructions, the A-stream has perfect knowledge of whether the dynamic sequence is effectual or ineffectual.

Our simulator models a wide-superscalar processor, SMT implemented on a wide-superscalar processor, and an SMP composed of superscalar processing elements. We implement the following hardware constraints and assumptions:

- The A-stream (or single thread in the superscalar processor) can fetch any number of *sequential* instructions in a single cycle, up to the maximum fetch bandwidth. The R-stream, because it has any number of non-contiguous PCs available from the A-stream, can fetch multiple *non-contiguous* basic blocks from the banked instruction cache (alternatively, *instructions* could be passed in the communication buffer as well). The latter is a key requirement for the R-stream to exploit A-stream value predictions and execute with peak parallelism.
- Instruction fetch, dispatch, issue, execute, and retire stages are modeled. Fetch and dispatch take 1 cycle each. Issue takes at least 1 cycle, possibly more if the instruction must stall for operands. Execution takes a fixed latency based on instruction type, plus any time spent waiting for a result bus. Address generation takes 1 cycle, and all data cache accesses are 1 cycle (i.e. perfect data cache). Instructions retire in order. Functional units are symmetric and fully

pipelined. Output and anti-dependences for both registers and memory are eliminated. Oracle memory disambiguation is used.

4.2.2 Results

In Figure 8, instructions retired per cycle (IPC) is given for 1) SS(64-4), a 4-way superscalar processor with a 64-entry reorder buffer, 2) an SMP composed of two SS(64-4) processing elements, 3) SS(128-8), an 8-way superscalar processor with a 128-entry reorder buffer, and 4) SMT implemented on SS(128-8). Of course, in the dual-threaded SMP/SMT, IPC is computed as the number of instructions in the original program divided by the number of cycles to complete.

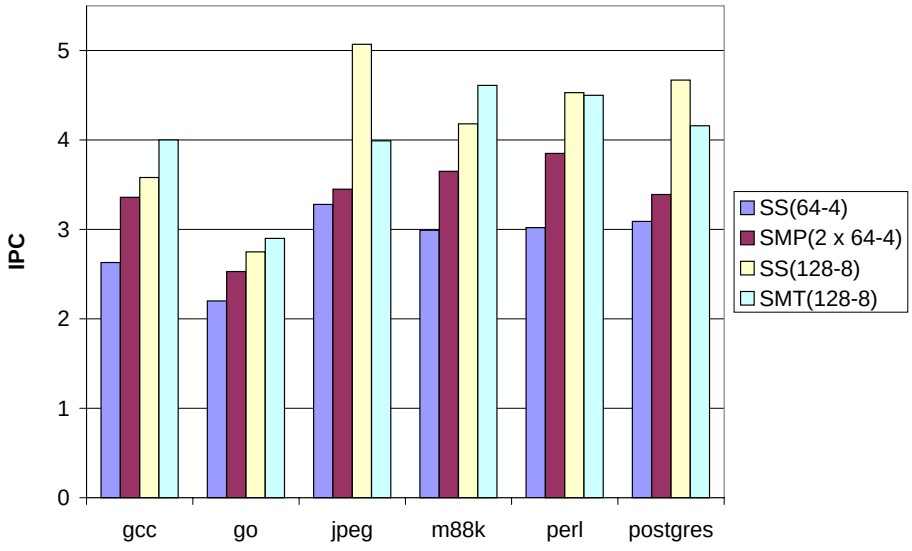

FIGURE 8. Performance comparison of superscalar and SMP/SMT exploiting ineffectual regions.

By adding a second processing element to SS(64-4) and exploiting IRs, the SMP improves performance across all of the benchmarks. Although doubling the complexity to SS(128-8) gives a larger improvement (the SMP is fundamentally limited to a throughput of 4 IPC), the SMP is an attractive alternative -- in terms of clock rate and flexibility. That is, overall performance (IPC *and* clock rate) and functionality potentially favor the SMP.

The SMT performs better than superscalar with the same underlying processor, SS(128-8), for half of the benchmarks. SMT overcomes the fundamental 4-IPC bottleneck of the SMP model. Poor utilization of 8-way issue by *gcc*, *go*, and *m88ksim* result in the redundant threads not being a performance problem, and IRs provide a net performance improvement. Performance is noticeably degraded for *jpeg*, however, because this benchmark exploits 8-way issue and because there is a lack of length-20+ IRs in *jpeg*.

5. Summary

A processor executes the full dynamic instruction stream in order to compute the final output of a program, yet we observed equivalent, smaller instruction streams that produce the same correct output. Unreferenced writes, writes that do not modify the value of a location, and correctly predicted branches all contribute significantly to the number of ineffectual instructions.

Based on these experiments, we identified large, dynamically-contiguous regions of instructions that are ineffectual as a whole: they either contain no writes, writes that are never referenced, or writes that do not modify the value of a location. Ineffectual regions (IRs) of length 20 instructions or more account for 15% to 40% of all ineffectual instructions in the program; and only 5-10% of all unique IRs account for most of these ineffectual instructions.

The architectural implication of this work is that instruction fetch/execution can quickly bypass predicted-ineffectual regions, while another thread of control verifies that the implied branch predictions in the region are correct and that the region is truly ineffectual.

6. Bibliography

[1] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. 23rd Intl. Symp. on Computer Architecture, pages 191-202, May 1996.

[2] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang. The Case for a Single-Chip Multiprocessor. 7th Intl. Symp. on Architectural Support for Programming Languages and Operating Systems, October 1996.
[3] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. 29th Fault-Tolerant Computing Symposium, June 1999.
[4] M. Lipasti, C. Wilkerson, and J. Shen. Value Locality and Load Value Prediction. 7th International Conf. on Arch. Support for Prog. Lang. and Operating Systems, October 1996.
[5] M. Lipasti. Value Locality and Speculative Execution. PhD Thesis, Carnegie Mellon University, April 1997.
[6] A. Sodani and G. S. Sohi. Dynamic Instruction Reuse. 24th International Symposium on Computer Architecture, June 1997.
[7] A. Sodani and G. S. Sohi. An Empirical Analysis of Instruction Repetition. 8th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1998.
[8] J. Huang and D. Lilja. Exploiting Basic Block Value Locality with Block Reuse. 5th Intl. Symp. on High-Perf. Computer Architecture, Jan 1999.
[9] A. González, J. Tubella, and C. Molina. Trace-Level Reuse. Intl. Conf. on Parallel Processing, September 1999.
[10] Y. Sazeides and J. E. Smith. Modeling Program Predictability. 25th Intl. Symp. on Computer Architecture, June-July 1998.
[11] S. McFarling. Combining Branch Predictors. Technical Report TN-36, WRL, June 1993.
[12] P. Chang, E. Hao, and Y. Patt. Target Prediction for Indirect Jumps. 24th Intl. Symp. on Computer Architecture, June 1997.
[13] D. Kaeli and P. Emma. Branch History Table Prediction of Moving Target Branches due to Subroutine Returns. 18th Intl. Symp. on Computer Architecture, May 1991.
[14] D. Burger, T. Austin, and S. Bennett. Evaluating Future Microprocessors: The Simplescalar Toolset. Tech. Rep. CS-TR-96-1308, CS Dept., Univ. of Wisconsin - Madison, July 1996.
[15] *PostgreSQL Global Development Team*. http://www.postgresql.org.
[16] V. Ercegovac and D. Goldman. ECE752 Class Project: Port of Postgres to Simplescalar. Univ. of Wisconsin - Madison, Spring 1999.
[17] *Transaction Processing Performance Council*. TPC-R Benchmark Specification, http://www.tpc.org/rspec.html.
[18] Q. Jacobson, E. Rotenberg, and J. Smith. Path-Based Next Trace Prediction. 30th International Symposium on Microarchitecture, pages 14–23, December 1997.
[19] S. Patel, D. Friendly, and Y. Patt. Evaluation of Design Options for the Trace Cache Fetch Mechanism. IEEE Trans. on Computers, 48(2):193–204, Feb 1999.
[20] E. Rotenberg, S. Bennett, and J. Smith. A Trace Cache Microarchitecture and Evaluation. IEEE Trans. on Computers, 48(2):111–120, Feb 1999.