McGill University
School of Computer Science
Sable Research Group

Fast Local List Scheduling

Sable Technical Report No. 2002-5

Clark Verbrugge
School of Computer Science
McGill University
clump@cs.mcgill.ca

July 23, 2002

www.sable.mcgill.ca

Abstract

List scheduling is a well-known $O(n^2)$ algorithm for performing instruction scheduling during compiler code generation on modern, superscalar architectures. An asymptotically more efficient $O(n \log(n))$ variation is presented; this algorithm achieves its lower time bounds by asserting practical constraints on the structure of instruction dependencies.

1 Introduction

List scheduling is a well-known algorithm for performing instruction scheduling during compiler code generation on modern, super-scalar architectures. The algorithm as originally designed has time complexity $O(n^2)$ in the number of input instructions [11]. In this paper an $O(n \log(n))$ variation on the list scheduling algorithm is described; this algorithm achieves its reduced time bound by sacrificing generality for practicality: by guaranteeing certain reasonable input restrictions the cost of constructing internal data structures is minimized, and the lower time bound results.

1.1 Motivation

Workstation processors now commonly make use of multiple execution units. The intent is that by allowing more than one instruction to be executed at the same time, an overall faster execution rate will ensue. In order to achieve these speedups, however, it is important that the instructions passed to the processor be ordered in a manner that facilitates exploitation of the specific architecture's execution model. Instruction scheduling thus becomes an important aspect of code generation within compilers directed at such architectures, whether the compiler is static or "just-in-time."

Unfortunately, scheduling can be a time-consuming phase of compilation. For static compilers this is often acceptable, but for JIT compilers where code generation is performed each and every time the program is executed, this can be prohibitively expensive. For instance, list scheduling with its $O(n^2)$ upper-bound was deemed too slow for IBM's JIT for Java [28]. Fast scheduling algorithms are clearly still quite important.

1.2 Improving Scheduling Complexity

The complexity of the list scheduling algorithm is dominated by the need to know whether two given instructions have any necessary ordering dependencies. Constructing and using the data structure, (the *data Dependence Directed Acycle Graph* or DDAG) that provides this information is a $\Theta(n^2)$ process—generally-speaking, each instruction must be compared to every other instruction in order to know if the instructions have a dependency. In the worst case, each of n input instructions could have a dependency on every other instruction.

Instruction dependencies, however, are typically not arbitrary or unconstrained, and an $O(n^2)$ dependency-checking algorithm is not always warranted. Most processors include a limited number of actual resources that can be the source of instruction dependencies: the number of registers, execution units, and buses for instance are all small and fixed. Memory access dependencies still remain theoretically unbounded, but in a practical sense there are few architectures that can efficiently accommodate massive, concurrent memory accesses. Moreover, with increased usage of indirect access (e.g., virtual function call setup in object-oriented languages), many memory accesses are unknown at compile-time. In either case, a coarse-grained view of memory can be sufficient for reasonably efficiency. Pragmatically then, a scheduling algorithm only needs to concern itself with an instruction ordering based on a limited, constant number of dependency-generating resources, and not necessarily with the full possibility of arbitrary, pair-wise instruction dependencies

This observation is key to reducing the time complexity of scheduling. Instead of comparing each instruction with every other, instructions are instead "bucketed" into lists associated with the resources they require. Associating the dependency information with the resource rather than the instruction allows the dependencies of an instruction to be determined in time proportional to the number of machine resources.

1.3 Overview of the Paper

Section 2 discusses relevant related approaches to instruction scheduling. Section 3 describes our algorithm; first an overview, then in Sections 3.1 and 3.2 the basic data structures and initial setup. Once the data structure is initialized, the schedule order of the instructions can be computed. This is done by first heuristically assigning priorities to the instructions in Section 3.3, and finally emitting the instructions in Section 3.4. In Sections 4 and 5 directions for future work and conclusions are discussed.

2 Related Work

Instruction scheduling has of course been well-examined in the literature, and the topic is usually covered in modern advanced compiler construction textbooks (*e.g.*, [20, 19, 29]). While most current efforts are focused on global scheduling [4, 6, 12, 13, 18, 26], there is a sizable body of literature on local scheduling techniques. General surveys of instruction scheduling include those by Smotherman *et al* [27], Krishnamurthy [15], and Ramakrishna Rau and Fisher [23].

List scheduling [11] is perhaps the most well-known algorithm for local instruction scheduling. Extensive work has been done improving this basic algorithm, primarily by adjusting the heuristic used for instruction selection [1, 27, 10]. Other local techniques include Balanced scheduling [14], randomized methods such as Backward and Iterative Repair scheduling [25, 7] and use of genetic algorithms [3], and many others. All of these algorithms are at least polynomial: they are aimed primarily at generating better code, not specifically reducing scheduling time.

An approach explicitly concerned with scheduling time is done by Leung, *et al* [17, 16]. The algorithms used, however, continue to be polynomial in the worst case—the time constraints are enforced by ensuring that partial results are valid. This allows the algorithm to be terminated if it exceeds a specific time span, while still reaping some benefit from the time spent.

Resource-based scheduling techniques bear more similarity to our approach. Reservation table algorithms, for instance, build a two-dimensional matrix explicitly modelling each instruction's resource requirements over time. Techniques have been developed to reduce the size of these tables [9, 22]. Finite automata approaches represent similar information, encoded as a finite automaton [21, 2]. Reservation tables and finite automata are used to construct efficient query mechanisms, quickly calculating the parameters of the resource dependencies of each pair of instructions. In contrast, our approach is more abstract, representing dependencies through a form of dependency graph that does not explicitly represent time. As well, our algorithm constitutes a complete instruction scheduling system, tracking dependencies as well as actually selecting instructions and emitting a schedule, all within specified asymptotic bounds.

3 Algorithm

The algorithm follows the basic functional pattern of list scheduling. Instructions are progressively inserted into a data structure representing the instruction dependencies. Once all instructions have been inserted, instructions that have had all their dependencies satisfied are removed from the data structure and become eligible to be scheduled in an order based on heuristic priorities. Detailed descriptions of list scheduling can be found in many compiler construction textbooks (see Section 2).

A key component of our algorithm is its efficient calculation and storage of dependency information. Instead of comparing each instruction with every other, instructions are instead “bucketed” into lists associated with each resource they require. Demands on each resource are further divided into *write* or *read*, representing exclusive and non-exclusive resource requirements respectively. Sections 3.1, 3.2 and 3.4.1 describe the organization and initial population of this critical data structure.

Once the data structure has been built, heuristic priorities are calculated for each instruction. Section 3.3 describes the implementation of a typical priority calculation, analogous to a bottom-up DDAG computation. Of course top-down and other priority calculation methods are also possible.

Instructions with all resources enabled are extracted from the data structure, and sorted by priority into a *schedulable* list. Instruction are then removed from this list for actual output/scheduling. Note that once scheduled, an instruction does not immediately disappear from consideration; rather, instruction latency implies that a resource may not be freed until some number of cycles after the instruction has been consumed. Resource tracking and revision of the data structure as instructions are consumed are detailed in Section 3.4.2.

3.1 Data Structures

The basic dependency data structure in our algorithm is an array of lists, whereas in list scheduling it is the DDAG. The difference is illustrated through the code on the left side of Figure 1. Here a sequence of three registers moves is followed by two writes to unknown memory locations; the associated DDAG is shown on the right. In this graph structure there are nodes for each instruction and directed edges between nodes representing dependencies. Instructions 1 and 4, for instance, must be executed prior to instruction 5 due to the flow-dependence on $r3$ and the possibility of an output-dependence in the write to memory respectively.

```
1) mov r3 ← r4
2) mov r7 ← r4
3) mov r4 ← r2
4) mov [..]  ← r4
5) mov [..]  ← r3
```

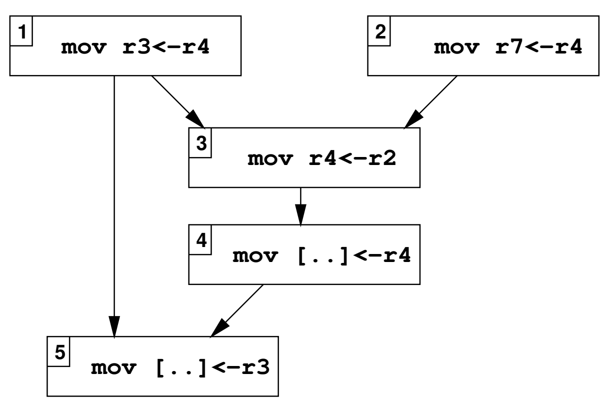

Figure 1: Example code and associated dependency graph.

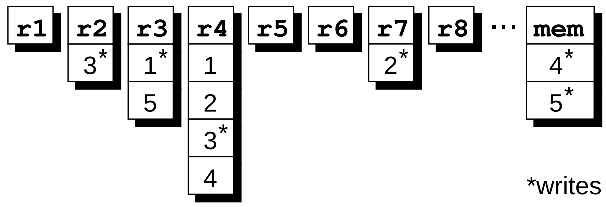

Figure 2: Array of Dependency Lists (ADL) structure for the code of Figure 1.

Figure 2 shows the same information represented with a different data structure, an *Array of Dependency Lists,* or ADL. Here each list is associated with a resource, either a register or the single generic category for unknown memory locations (more than one such category is possible). Within each resource list, instructions dependent on that resource are listed in the order in which they were originally added to the ADL; reads and writes are also distinguished. That instruction 5 depends on instructions 1 and 4 is then represented by two separate entries: first, a read entry for instruction 5 below the write entry for 1 within the $r3$ list, and second a write entry for instruction 5 below the write entry for 4 in the *mem* list.

Since it will be necessary to locate all dependencies of an instruction, each instruction maintains a singly-linked list of its entries in the ADL, and symmetrically each entry in the dependency list points back at its owning instruction. Figure 3 shows the data structure schematic, this time generalized to allow more than one memory class (or perhaps other dependency-generating resources).

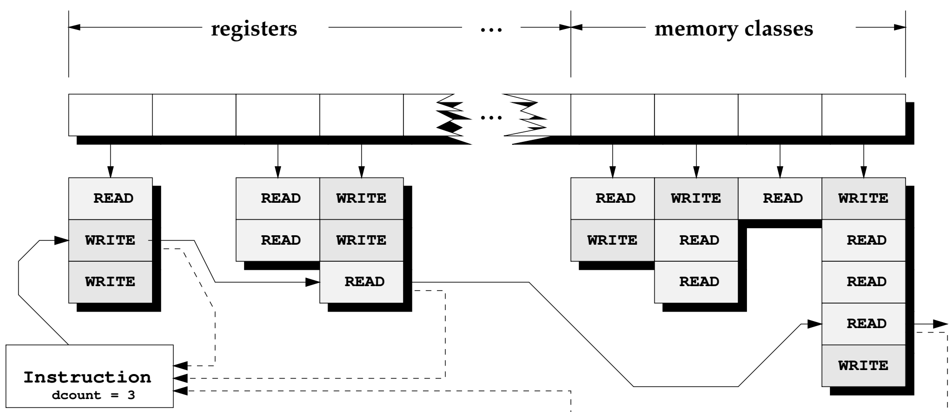

Figure 3: Associating dependencies with instructions.

3.2 Algorithm Setup

The algorithm can be divided into three main phases: first, instruction entries are stored into the ADL as they are fed to the scheduler; second, priorities are assigned to the instructions, and third, instruction entries are extracted from the ADL to generate a scheduling order. Of these the first two are relatively simple linear operations; the majority of the work is done in the last stage, and is the source of overall complexity of the algorithm.

3.2.1 Building the ADL

Inserting instructions into the ADL is trivial; as instructions are issued to the scheduler an entry for each instruction is added to the end of the list for each possible resource the instruction may require. Links between entries and instructions (as described in Figure 3) also need to be established. Additional information stored within each entry includes whether an entry represents a read or a write dependency, and how long it monopolizes the resource (the latency).

Some heuristic priority schemes require bottom-up traversals of the dependency data structure. In order to allow that to be done quickly, extra effort is required at insertion time, and an extra *child* pointer is located in every entry. As each entry `e` is made within a list, `e.child` is pointed at `null` and the prior entry in the same list, call it `p`, is examined. The situation will fall into one of four categories depending on whether `p` and `e` are read or write entries. Figure 4 illustrates the transitions, which are handled as follows:

1. Cases *A* and *B*: `p` is a write. Then `p.child` is redirected to `e`.
2. Case *C*: `p` is a read and `e` is a read. The `child` pointer of `p` is not changed.
3. Case *D*: `p` is a read and `e` is a write. Then the `child` pointers of `p` and the entire sequence of *contiguous* prior reads in the same list from `p` up are all redirected to `e`

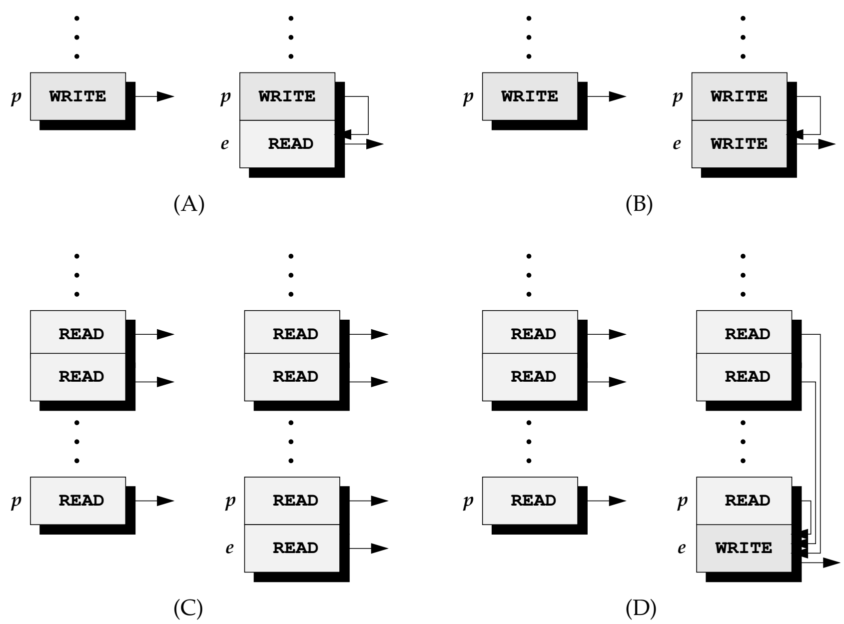

Figure 4: Adjusting the child pointers when inserting dependencies.

REMARKS: Each list is essentially maintaining a dependency graph with respect to that resource; instructions within are ordered in their dependency order, with the implicit understanding that sequences of reads may be executed in an arbitrary order. The `child` pointers describe this structure; each `child` pointer of an entry `e` points to either `null`, in which case `e` is a leaf in the dependency graph, the next instruction if `e` is a non-leaf write, or finally if `e` is a non-leaf read then `e` points to the next write instruction in the same list; an example dependency-graph style depiction of the same information in a list structure is shown in Figure 5. The operations described in Figure 4 preserve this invariant.

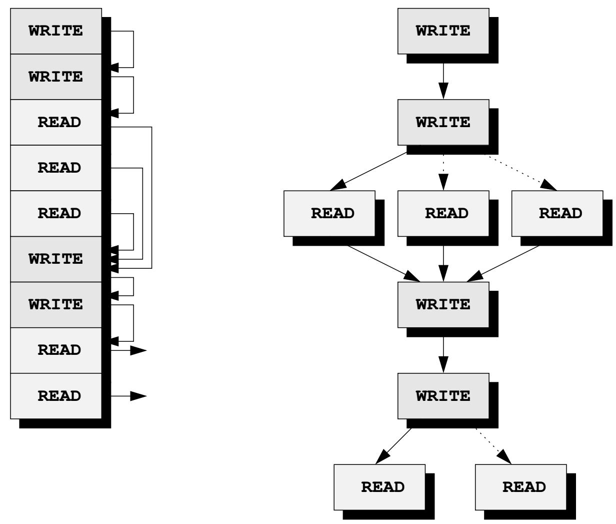

Figure 5: Equivalent list and graph dependency structures. The ADL column on the left represents the same dependency information as the dependency graph on the right, for one resource. Since the ADL list includes only a single `child` pointer for each entry, contiguous sequences of reads are interpreted as all children of the same prior write. The dotted arrows in the ADL list figure indicate parent/children relations that are implicit in this scheme.

Notice that in Figure 5 the dependence graph equivalent of the list representation appears as a sequence of levels, consisting of either a single write or a contiguous section of reads. ADL entries in different levels are actually dependent, whereas ADL entries in the same level do not imply a dependence. This motivates the following definition.

Definition 3.1 *A* stratum *is a set of ADL entries from a single list, consisting of either just one write entry, or a contiguous sequence of read entries such that no immediately prior or immediately subsequent entry to that sequence is also a read. Every ADL list can be decomposed into exactly one ordered (top to bottom) series of* strata.

3.3 Determining Priorities

As with list scheduling, many heuristic priority assignment schemes are possible. Here we describe how to implement the simple "longest distance to a leaf" priority; this would be calculated on a DDAG as the maximum priority of any child node (a recursive calculation) plus the execution time of the instruction.

Figure 6 illustrates the data structure setup for doing this with on the ADL: an index array separate from the ADL itself is used to point (initially) to the end of each list. These indices will be progressively moved up each list, maintaining a representation of the boundary of a bottom-up (post-order) traversal of the data structure; the process is summarized in the (very) pseudo-code of Figures 7, 8 and 9.

Figure 7 illustrates the main loop. While any instructions remain to be prioritized, the entire index list for the ADL is repeatedly inspected. Each iteration, the element of each list indicated by the index array is examined. If this element, e, has not been seen before, it and every entry in that same stratum is marked seen.

The actual `mark_seen` function is shown in Figure 8. This function is called every time an ADL en-

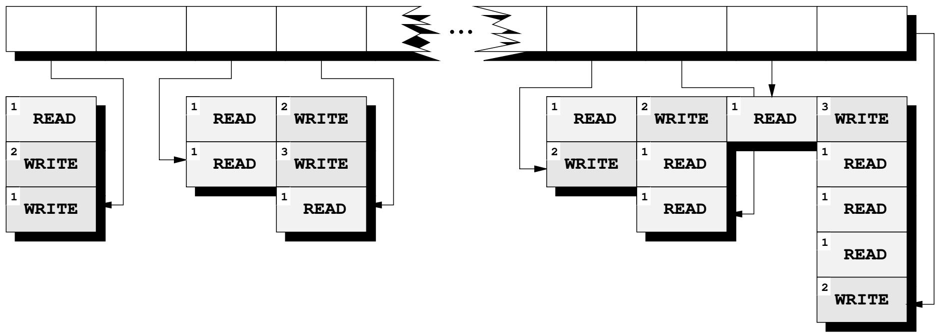

Figure 6: Calculating priorities of instructions. Numbers in the upper-left of each entry are latencies.

```
priorities():
   while index_array is not all null entries
      for each list l in the ADL
         if index_array[l] ≥ 0
            let e = ADL[l]→elements[index_array[l]]
            if e →seen == false
               let R = every entry in the same stratum as e
               for each r ∈ R
                  r →seen = true
                  mark_seen(r)
               end for
            end if
         end if
   end while
```

Figure 7: Main loop for calculating priorities.

```
mark_seen(e):
   let E be the ADL entries of e →instruction
   if all d ∈ E have been seen
      calc_priority(e →instruction)
      for all d ∈ E
         let l_d be the list for entry d
         --(index_array[l_d])
      end for
   end if
```

Figure 8: Marking an instruction.

try is first encountered in the main loop; if this means that all ADL entries for the associated instruction (γ) have now been seen, then the actual priority of γ can be calculated, and the traversal boundary indicated by the `index_array` needs to be updated. This latter step is accomplished by decrementing the index of the `index_array` entry associated with the lists of each ADL entry for γ. If a particular ADL entry (d) of γ is a write, then d is guaranteed to be the same entry indicated by `ADL[`l_d`]->elements[index_array[`l_d`]]`, and decrementing `index_array[`l_d`]` will simply move our traversal boundary up past that entry. If it is a read, then d may be the `index_array[`l_d`]`'th entry or any prior read entry in the same list (any one of the same stratum). Since all reads in a stratum are marked seen at the same time, their individual identities are no longer important: none will be marked again, so as long as the same number of decrements of `index_array[`l_d`]` occur as reads were marked, the traversal boundary will not move upward until all are eliminated. With reference to the right-hand side of Figure 5, this is equivalent to removing a leaf element from the dependency graph; if it is a write then this exposes a prior stratum of the graph, but if it is a read then all reads in that same stratum have to be deleted (in any order) before the prior stratum becomes visible.

```
calc_priority(γ):
   let E be the ADL entries of γ
   let maxpriority = 0
   for all d ∈ E
      if d →child != null
         let C = every entry in the same stratum as d →child
         for each c ∈ C
            let λ = c →instruction
            maxpriority = max(maxpriority,λ →priority)
         end for
      end if
   end if
   γ →priority = maxpriority + γ →latency
```

Figure 9: Calculating priority for an individual instruction.

The remaining portion of the algorithm is the code to calculate the priority of an individual instruction, shown in Figure 9. Our heuristic is to calculate priority as the maximum priority of any dependent instruction plus the instruction's own latency. In `calc_priority` all ADL entries d of the instruction are considered. For each such d, the stratum of d`.child` is examined; in the right-hand side of Figure 5 this would be equivalent to looking at the set of entries forming the next stratum down—all children of d. Priority of γ is then calculated as the maximum priority of any of these child entries, plus of course the latency of γ.

3.4 Scheduling the Instructions

Once the ADL structure is built and priorities assigned, actual scheduling can occur. This involves first identifying the initial set of schedulable instructions, and then successively scheduling instructions until all instructions have been scheduled.

3.4.1 The Initial Schedulable Set

For determining schedulable instructions, a setup similar to that required for the priority assignment is used (Figure 10 illustrates the structure). An index array is built, but this time every entry of it points to the *first* entry of each list; this index array will indicate the frontier of ADL entries that have not yet been completely processed by the scheduler.

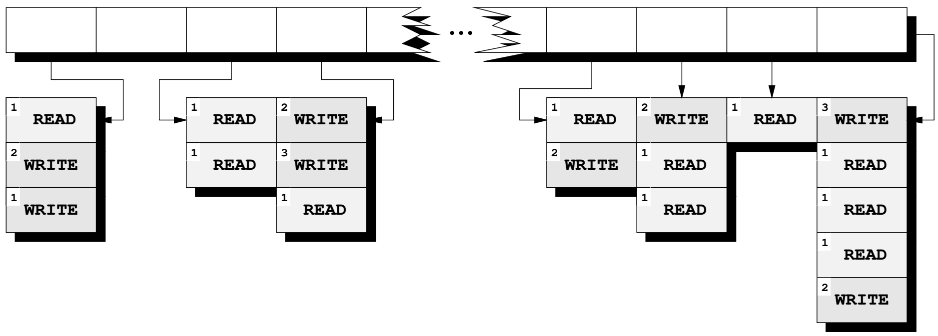

Figure 10: Data structure setup for actual scheduling.

Figure 11 gives pseudo-code for constructing the initial set of schedulable instructions given the index array data structure setup. Each instruction is assumed to have data fields for recording (and initialized to) the total number of ADL entries for that instruction (`dep_total`), as well as a counter (`dep_count`) from 0 to that number.

```
first_schedulable():
   for each list l in the ADL
      if index_array[l] ≥ 0
         let e = ADL[l]→elements[index_array[l]]
         let R = every entry in the same stratum as e
         for each r ∈ R
            let γ = r →instruction
            if ++(γ →dep_count) == γ →dep_total
               add γ to the schedulable list
            end if
         end for
      end if
   end for
   sort schedulable list by priority
```

Figure 11: Generating the set of instructions that are initially schedulable.

A single pass is made over the ADL, and the entries within the top stratum of each list are examined. Note that being in the top stratum of each list, these ADL entries represent the resource requirements that are not dependent on any other instruction. As each entry is examined, the `dep_count` field of the associated instruction is incremented. Once this value reaches the `dep_total` value, all ADL entries for that instruction have been seen, and thus all required resources are available: the instruction is a candidate for scheduling.

The schedulable list now contains the pool of instructions with no dependencies. This then has to be sorted by priority; simple algorithms will suffice in practice, since this list is unlikely to be large. For theoretical purposes, however, note that with a constant upper bound λ on the latency of any instruction and our particular priority heuristic, there is a consequent maximum priority of $n\lambda$. This means instructions can be radix sorted (bucketed) based on their priority into an array of size $n\lambda$ in linear time. For scheduling, it is not necessary to individuate available instructions with the same priority, so a simple pass through this array aggregating bucket lists will sort the initially-schedulable list in $O(n)$ time.

3.4.2 Determining the Schedule Order

Actually scheduling involves successively extracting the highest priority instruction from the schedulable list as the next in the schedule order and updating the data structure.

Emitting the highest priority instruction is a simple task: the head of the (sorted) *schedulable* list is removed and becomes the next scheduled instruction. At the same time, this instruction is added to a *scheduled* set, a set consisting of all instructions that have already been emitted, but whose latencies have not all elapsed (an instruction with no latencies need not be added). Figure 12 gives the pseudo-code.

```
emit_schedule():
   while instructions remain to be scheduled
      do
         process_scheduled()
         let γ = remove_head(schedulable)
      while γ == null
      emit γ
      add γ to the scheduled set
   end while
```

Figure 12: Emitting the next schedulable instruction.

Note that each iteration of the loop in Figure 12, and as long as the *schedulable* list is empty, the `process_scheduled` function is called to update the data structure. This function has the task of filling the *schedulable* list after the `first_schedulable` function has initialized it. Pseudo-code is shown in Figure 13.

The `process_scheduled` function begins by initializing the *now_schedulable* list to be empty; this list will hold the instructions this function determines need to be added to the *schedulable* list. Next, it iterates over the *scheduled* list. For each such instruction, it decrements the latencies of all ADL entries with a non-negative latency. If an entry's latency is reduced to 0 or less, the `index_array` used to delineate the frontier of processed ADL entries is incremented, and if that exposes a new stratum, the `expose_stratum` function is called to handle the newly-available resource.

A second pass is then made through the ADL entries of the given instruction to see if the latencies of all associated ADL entries have been decremented below 0. If they have, then the instruction has completely exited the scheduler's consideration, and it is deleted from the *scheduled* list. Finally, the instructions that have become schedulable during this function call are added to the main *schedulable* list.

The `expose_stratum` function is shown in Figure 14, and is the function that will actually determine whether an instruction needs to be added to the *schedulable* list. This is called with the first entry in a newly-exposed stratum. Each entry in that stratum is then examined, and the `dep_count` of the associated instruc-

```
process_scheduled():
   set the now_schedulable list to empty
   for each instruction γ in the scheduled list
      for each ADL entry of γ, e
         if e →latency ≥ 0 and --(e →latency) ≤ 0
            e →latency = -1
            let l_e be the list containing e
            inc index_array[l_e]
            let d = ADL[l_e]→elements[index_array[l_e]]
            if d and e belong to different strata
               expose_stratum(d)
            end if
         end if
      end for
      if all e ∈ E have e →latency < 0
         remove γ from the scheduled list
      end if
   end for
   add all entries of now_schedulable to schedulable
```

Figure 13: Processing the *scheduled* list.

```
expose_stratum(a):
   for each e in the same stratum as a
      let γ = e →instruction
      if ++(γ →dep_count) == γ →dep_total
            add γ to the now_schedulable list
      end if
   end for
```

Figure 14: Exposing the next stratum within a newly-freed resource.

tion incremented. This is a continuation of the process described with respect to the `first_schedulable` function (Figure 11). Once all ADL entries for an instruction have been accounted for by that count, the instruction is ready to be scheduled, and is added to the *now_schedulable* list.

3.4.3 Analysis of Scheduling

Correctness can be established by arguing that no instruction is scheduled before its required resources are available, and all flow/anti/output dependencies are appropriately respected. A formal proof is not presented here.

A more interesting aspect of the analysis of this algorithm is the time complexity. A constant-bounded number of resources for each instruction implies a linear number of ADL entries, and so an $O(n)$ upper time bound for the initial population of the data structure. Most of the actions of the *process_scheduled* function preserve this bound: the cost of examining ADL entries, updating the frontier of processed entries, adding instructions to the *now_schedulable*, and processing the *scheduled* list can all be amortized to a constant cost per instruction.

However, each invocation of `process_scheduled` concludes with the directive to merge the entries of *now_schedulable* into the existing *schedulable* list. Since the latter list is assumed to be ordered by priority, this adds to our algorithm the complexity of merging an unordered list into a sorted list. The number of instructions in the *schedulable* and *now_schedulable* lists are not necessarily bounded by any small number, and this results in a general $O(n \log(n))$ time bound.

Proposition 3.1 *Given a constant-bound on the number of resources and maximum latency, the* emit_schedule *procedure has time complexity* $O(n \log(n))$.

Proof: (Sketch) All steps other than merging *now_schedulable* into *schedulable* can be performed in overall linear time. The latter step can be done in amortized time $O(n \log(n))$ by using a mergeable heap data structure, like a Binomial or Fibonacci heap [8], which has an amortized $O(n \log(n))$ cost over a sequence of heap create, insert, and extract-minimum operations. $\square$

4 Future Work

We have shown an improved theoretical complexity. It remains to be seen how this algorithm performs in practice. Actual measurments of course will require instrumentation of an existing compiler, or development of an appropriate test harness.

An important practical aspect of our algorithm is the limitation on the number of resources—long sequences of provably-distinct memory accesses are expected to be uncommon. Register saving on the stack prior to function calls may be one place that such situations do occur; however, since such complete saves are expensive in their own respect, reducing or eliminating saves is an important topic in interprocedural register allocation [5, 24]. We also point out that for popular architectures (eg, current PowerPC or Intel processors), a long sequence of memory accesses can easily overwhelm any internal buffering, and so is unlikely to be efficient under any schedule. Nevertheless, an experimental analysis of actual resource requirements would enable the time bounds of our algorithm to be appropriately refined.

5 Conclusion

Local scheduling is not the primary focus of current research aiming to maximally exploit instruction level parallelism. Demonstrably better results can be obtained by increasing the scope of the scheduler beyond basic blocks, and so current research concentrates almost exclusively on larger-scope scheduling algorithms. Nevertheless, local scheduling continues to have relevance to compilers with limited time resources available for scheduling: global and interprocedural scheduling are expensive operations that current Just in Time compilers cannot always afford.

The algorithm presented here has the advantage of being quick in theory, and it is expected in practice as well: all operations require straightforward algorithms and relatively simple data structures. Of course verification of the pragmatic tradeoffs we exploit to achieve the lower theoretical bound is still desired, and is the subject of future work.

References

[1] B. Appelbe, R. Das, and R. Harmon. Instructions scheduling for highly super-scalar processors. Technical Report GIT-CC-97-34, Georgia Institute of Technology. College of Computing, 1997.

[2] V. Bala and N. Rubin. Efficient instruction scheduling using finite state automata. In *Proceedings of the 28th International Symposium on Microarchitecture*, November 1995.

[3] S. J. Beaty. Genetic algorithms for instruction sequencing and scheduling. In *Workshop on Computer Architecture Technology and Formalism for Computer Science Research and Applications*, Apr. 1992.

[4] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. *ACM SIGPLAN Notices*, 26(6):241–255, June 1991.

[5] F. Chow. Minimizing register usage penalty at procedure calls. In *Proceedings of the SIGPLAN'88 Symposium on Programming Language Design and Implementation*, pages 85–94, 1988.

[6] K. D. Cooper and P. J. Schielke. Non-local instruction scheduling with limited code growth. *Lecture Notes in Computer Science*, 1474:193–, 1998.

[7] K. D. Cooper, P. J. Schielke, and D. Subramanian. An experimental evaluation of list scheduling. Technical Report TR98-326, Rice University, Department of Computer Science, Sept. 1998.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[9] A. E. Eichenberger and E. S. Davidson. A reduced multipipeline machine description that preserves scheduling constraints. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 12–22, May 1996.

[10] M. A. Ertl and A. Krall. Instruction scheduling for complex pipelines. In *Compiler Construction (CC'92)*. Springer Verlag LNCS 641, 1992.

[11] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. *SIGPLAN Notices*, 21(7):11–16, July 1986.

[12] M. Golumbic and V. Rainish. Instruction scheduling beyond basic blocks. *IBM Journal of Research and Development*, 34(1):93–97, January 1990.

[13] R. Gupta. A code motion framework for global instruction scheduling. In K. Koskimies, editor, *Compiler Construction (CC'98)*, pages 219–233, Lisbon, 1998. Springer LNCS 1383.

[14] D. R. Kerns and S. J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. *SIGPLAN Notices*, 28(7):278–289, July 1993.

[15] S. M. Krishnamurthy. A brief survey of papers on scheduling for pipelined processors. *SIGPLAN Notices*, 25(7):97–106, July 1990.

[16] A. Leung, K. Palem, and C. Ungureanu. Run-time versus compile-time instruction scheduling in superscalar (RISC) processors: Performance and tradeoffs. Technical Report TR1995-699, New York University, July 1995.

[17] A. Leung, K. V. Palem, and A. Pnueli. Highly efficient instruction scheduling of realtime programs on RISC processors. Technical Report TR1996-723, New York University, July 1996.

[18] U. Mahadevan and S. Ramakrishnan. Instruction scheduling over regions: A framework for scheduling across basic blocks. *Lecture Notes in Computer Science*, 786:419–, 1994.

[19] R. Morgan. *Building an Optimizing Compiler*. Digital Press, 1998.

[20] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, California, 1997.

[21] T. Muller. Employing finite automata for resource scheduling. In *Proceedings of the 26th International Symposium on Microarchitecture*, December 1993.

[22] V. J. Ramanan and R. Govindarajan. Resource usage models for instruction scheduling: Two new models and a classification. In *ACM SIGARCH International Conference on Supercomputing*, June 1999.

[23] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview, and perspective. *Journal of Supercomputing*, 7:9–50, July 1993.

[24] V. Santhanam and D. Odnert. Register allocation across procedure and module boundaries. In *Proceedings of the SIGPLAN'90 Symposium on Programming Language Design and Implementation*, pages 28–39, 1990.

[25] P. Schielke. Issues in instruction scheduling. Technical Report TR98-323, Rice University, Department of Computer Science, Sept. 1998.

[26] S. Schmidt. Global instruction scheduling — A practical approach. *Lecture Notes in Computer Science*, 1124:815–??, 1996.

[27] M. Smotherman, S. Krishnamurthy, P. Arvind, and D. Hunnicutt. Efficient DAG construction and heuristic calculation for instruction scheduling. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 93–102, Albuquerque, NM, November 1991.

[28] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java just-in-time compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[29] R. Wilhelm and D. Maurer. *Compiler Design*. Addison-Wesley, 1995.