Reducing Code Size in VLIW Instruction Scheduling

Steve Haga, Yi Zhang, Andrew Webber, Rajeev Barua

Department of Electrical & Computer Engineering

University of Maryland

College Park, MD 20742, U.S.A

Abstract

Code size is an important concern in embedded systems. VLIW architectures are popular for embedded systems, but often increase code size, by requiring NOPs to be inserted into the code to satisfy instruction placement constraints. Existing VLIW instruction schedulers target run-time but not code size. Indeed, current schedulers often increase code size, by generating compensation copies of instructions when moving them across basic block boundaries. Our approach, for the first time, uses the power of scheduling instructions across blocks to reduce code size and not just run-time, for a certain class of VLIWs. We therefore show that trace scheduling, previously synonymous with increased code size, can in fact be used to reduce code size on such VLIWs. Our scheduler uses a cost-model driven, back-tracking approach that starts with an optimal algorithm for searching the solution space in exponential time, but then also employs branch-and-bound techniques and non-optimal heuristics to keep the compile time reasonable (within a factor of 2). Our method reduces the code size for our benchmarks by 16% versus the best existing across-block scheduler, while being within 0.8% of its run-time.

1 Introduction

Very Long Instruction Word architectures (VLIWs) are the predominant design for high performance embedded systems because they offer comparable performance to the superscalars used in most desktops, but at a lower cost [10, 25]. The drawback of VLIWs is that they rely more heavily upon good compiler technology. The compiler must identify independent instructions for execution in parallel, and then place them together into fixed-length instructions groups called *long words*.

Instruction scheduling consists of two basic phases: 1) *across-block* and 2) *within-trace*. In the first phase, trace scheduling [11] or its later improvements [16, 22] derive *traces* – sequences of basic blocks with a high probability of following one after another. This phase also provides a *migration mechanism* for moving instructions between the basic blocks of a trace. The second phase actually schedules the instructions of the trace, utilizing the migration mechanism from the first phase to move instructions between blocks when beneficial. The second phase also considers the instruction-type restrictions per VLIW issue slot and the instruction dependencies. The problem of finding the best schedule is NP-complete [23] and so heuristics are used. Existing second-phase methods for scheduling within a trace

include list scheduling [23], finite state automata [3], and dynamic programming [18]. Most second phase schedulers use one-pass (greedy) heuristics, and yet are fairly effective at finding a fast schedule.

A significant drawback of existing scheduling methods is that they only aim to reduce the run-time of the executable, while ignoring the code size impact. In the across-block phase, the migration mechanism requires instructions to be duplicated, resulting in code bloat. In the within-trace phase, identifying a fast schedule does not necessarily imply that it is a compact schedule. Instead, there are often many available schedules, each equally fast but having different code sizes.

Existing methods ignore code size because they were designed either for older VLIWs where our methods would not apply, or for desktop processors where code size is not important. In embedded systems, however, code size is very important [6] for reasons of cost – code is usually stored in Read-Only-Memory (ROM), and smaller code implies a smaller (*i.e.*, cheaper) ROM that uses less power. Like existing approaches, our algorithm has an *across-block* phase and a *within-trace* phase. But unlike today's approaches, our methods reduce code size while *maintaining the run-time* of current techniques.

Our across-block phase's migration mechanism differs from those now used. Since moving instructions across blocks may cause code bloat, we choose to restrict such moves to cases that do not increase code size, either because duplication is not needed or because the duplicate instruction is able to replace an existing NOP (and therefore does not increase the code size). In addition, since our new restrictions may harm performance, we further consider lifting this restriction for the most-frequent traces.

Our within-trace phase also differs from existing run-time-oriented schedulers. Rather than modify a current, greedy scheduler so that it optimizes for code size, we instead develop a new, back-tracking scheduler that specifically optimizes for *both* code size and run-time. A back-tracking scheduler can undo earlier scheduling decisions when these are later found to be sub-optimal. Our back-tracking scheduler is based on an exhaustive search algorithm of exponential compile-time, but through aggressive and novel branch-and-bound pruning techniques, coupled with some non-optimal heuristics, we can *bound the maximum* compile time to a user-controlled value (currently 6 times the compile time using existing methods). The actual compile time is even smaller (currently within a factor of 2).

The intellectual novelty of our scheme is seen in the following three new contributions. First, although the idea of using a back-tracking scheduler is not new, we are the first to develop a back-

tracking technique targeted for code size, and to develop a series of innovative pruning strategies unique to a search for a minimum-code-size solution. Second, using across-block motion to reduce code size is novel. Existing across-block approaches generally increase code size. Our method constrains the within-trace scheduler to not move certain instructions across basic blocks if that move would likely increase code size. This is unlike all existing schedulers which do not need to place additional constraints on movement since the within-trace scheduler alone decides whether it is profitable to move instructions, based solely on run-time. Third, our method is unique in targeting different objectives for traces of different frequencies – code size only for infrequent traces; run-time for frequent traces; and both code size and run-time for traces of intermediate frequency. Such customization to differing objectives is not used in traditional schedulers because their objective is only to reduce run-time.

Compared to existing approaches, our scheduler reduces the code size by 15.8% with only a cost of 0.82% in run-time. This code size improvement can be divided among three sources. First, using only our within-trace scheduler improves code size by 8.8%. Second, our migration restrictions improve the code size by another 4.9%. Third, using a hybrid strategy that only targets code size for very *infrequent* traces and that removes the migration restrictions for very *frequent* traces improves code size by yet another 2.1%. ($8.8\% + 4.9\% + 2.1\% = 15.8\%$).

An outline of the rest of the paper is as follows. Section 2 describes which VLIWs can benefit from our method. Section 3 shows an example that demonstrates how minimum run-time solutions do not necessarily have minimum code size. Section 4 describes related work. Section 5 describes our backtracking scheduler as it applies to basic blocks, without considering special issues relating to moving instructions across basic blocks. Section 6 describes how the backtracking scheduler can be modified to run on an entire trace. Section 7 presents our hybrid strategy to do even better by changing the optimization criteria at the extremes. Section 8 lists experimental results. Section 9 concludes.

2 Applicable VLIWs

The specific VLIW architecture impacts the achievable benefit of our method in two ways. First, since we improve code size by reducing the number of NOPs in the code, architectures that tend to need more NOPs will provide us with more opportunities for reducing them. Below, we also consider those machines that provide an ISA mechanism to compress most of their NOPs. Second, since our within-

trace algorithm works by separating computation from stall cycles, the underlying hardware's stall mechanism affects the applicability of our within-trace approach. In particular, older VLIWs that use long-words filled with NOPs to specify stalls cannot benefit from our within-trace methods. A fuller description of hardware mechanisms for stalling and for NOP compression follows.

The only type of stall mechanism that does *not* benefit from our within-trace methods (but can still use our across-block methods) is when long words full of NOPs are used to specify stalls. To understand this mechanism further, consider that the simplest VLIW hardware does not provide any mechanism for stalling. Therefore, the compiler must analyzing the data dependencies to identify where stall cycles are needed and then insert long words filled with NOPs into the code at these points. In terms of code size, these NOP-filled long words are an expensive way to achieve a stall. In such systems, there is little the within-trace scheduler can do to reduce code size. For example, if a particular trace requires X cycles to execute, it will necessarily have a code size of X long words and cannot be made smaller. Because of its high code size cost, modern VLIWs rarely use this stalling method.

VLIWs employing any other mechanism for stalls will benefit (to varying degrees) from our within-traces methods. To understand why, let us consider the other potential ways to accomplish a stall. There are four stall mechanisms used in modern VLIWs: 1) hardware dependency checks, 2) stall bits, 3) multi-NOPs, and 4) using variable-width long words for stalls. First, in machines with hardware dependency checks, the hardware detects when to stall by checking if any input operand in the long word is not ready. An example is IA64 [28]. IA64 is not an embedded processor, but similar designs have been proposed for embedded systems, such as TEPIC[19]. Second, in machines with stall bits, the number of cycles to stall is encoded into special bits in every long word. Such a method is described in [1, 13]. Third, in multi-NOP machines the NOP opcode contains an argument field that specifies the number of cycles to stall after executing the current long word. Thus, by scheduling a single NOP in the last cycle before the stall, empty long words are avoided in the code. An important example of such a processor is the Texas Instruments TMS320C6x [31]; an older one is [26]. The TMS320C6200 and C6400 use NOPs not only to separate long words and perform stalls, but also to align packets. Fourth, some VLIWs may use their variable-width feature to achieve stalls at a reduced code-size cost. Although NOPs must be inserted into the code for every stall cycle, the variable width of long

words, means that only one NOP is needed per cycle, as compared to a simple VLIW that inserts a full long-word of NOPs. We note that not all VLIWs with variable-width long words necessarily use this mechanism to achieve stalls; multi-NOPs and EPICs are two examples of variable width-machines that have already been mentioned as providing a more sophisticated stall mechanism. One example of a machine that does use this feature for stalls is Infineon's Carmel[35].

In all four above classes of VLIWs, long words filled with NOPs are not used to specify stalls, but NOPs still occur for a variety of reasons. Some other sources of NOPs affecting one or more of the above classes of machines are: alignment requirements (such as for the start of basic blocks or for long words that cross cache-block boundaries), a lack of sufficient parallelism, instruction issue restrictions, stalls that require single NOP slots, and multi-NOPs. Our method reduces all of these sources of NOPs, in all four of these classes of VLIWs. Specific pruning strategies may need to be modified to target the unique features of some of these architectures, however. For example, Carmel contains additional NOPs when defining CLIWs, which are Custom-Long-Instruction-Words that are dynamically declared and reside in a special on-chip table. Once defined, they can be executed with special table-look-up operations. Using these CLIWs Carmel is able to reduce the code size and run-time of frequent code. The proper scheduling of these CLIWs can reduce not only the code size of their declarations, but also contention in the small hardware table that stores the CLIWs.

For other VLIWs, the impact our technology is less. In particular, some VLIWs compress the NOPs in the long words by setting special bits within the instruction.[17]. Since the NOPs do not appear directly in the binary, these machines have a variable-length-execution-set (VLES). To some extent, the code-size benefit of avoiding NOPs is offset by the fixed code size cost of special encoding bits in the ISA (and also by certain other costs such as an extra pipeline stage for unpacking long words). Examples of VLES machines are the Philips Trimedia processor [32] and Tiger Sharc [33]. In such architectures, non-encoded NOPs are generally only needed for alignment problems. As a result, since these machines have so few non-encoded NOPs to begin with, reducing them further does not improved the code size. Our methods can still have benefit for many of these machines, as the NOPs are often inserted back into the long words when the instructions enter the I-cache, *e.g.,* Phillips Trimedia. In such a case, our approach reduces to a technique for shrinking the footprint of the code in the I-cache.

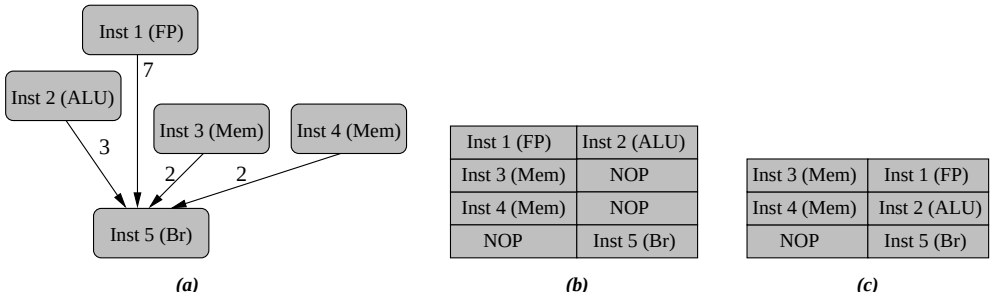


Figure 1: VLIW scheduling example. (a) DFG; (b) result of list scheduling, assuming that only one slot can perform memory operations; (c) an optimal schedule.

3 Minimizing code size and run-time

Here we present an example of how a current-day scheduler can produce a schedule with minimum run-time, but non-minimum code size. This example is important since it underscores the difference between a method that successfully produces a minimum run-time solution and one that also yields the smallest code size. Figure 1(a) presents the dataflow graph (DFG) for the dependencies of a sample basic block. The nodes are instructions; the edges are dependencies between instructions; the numbers attached to the edges are their latencies. Let us try to schedule this DFG on a simple 2-wide VLIW that restricts memory instructions to only the first slot.

Figure 1(b) shows the run-time schedule that results from list scheduling [23]. Although only four long words are scheduled in 1(b), the run-time is eight cycles, which is, incidentally, optimal in this case, since the critical path in the DFG is also eight cycles. (For simplicity of presentation, this example assumes the presence of hardware stalls. Similar examples can be easily constructed for multi-NOP systems.) While the schedule of 1(b) has achieved the optimal run-time, its code size is not minimum. Figure 1(c) has the same run-time (eight cycles), but a smaller code size (three long words instead of four). This example shows how current methods that optimize for run-time alone may not produce minimum code size solutions. Fortunately, it also shows that minimizing code size need not sacrifice run-time – our scheduler minimizes code size but retains the same run-time. In fact, there typically exists an entire family of optimal-run-time schedules for a basic block, and these schedules may have different code sizes. This example uses list scheduling because it is easy to understand. Better methods than list scheduling may occasionally find a faster schedule, but they do no better in terms of code size because *all existing methods only consider run-time*, so that Figures 1(b) and 1(c) *appear equally good.*

Delving deeper, this example illustrates a fundamental drawback of all greedy schedulers in use today. In Figure 1(b), the greedy scheduler, choosing instructions based on the earliest deadline, places instructions #1 and #2 in the first long word. This is optimal so far, but *it does not consider the impact of the current scheduling choice upon the remaining code*. In a backtracking scheduler such as ours, this mistake of not reducing contention for the memory slot can be undone, and other alternatives explored.

4 Related work

As we have seen, the first phase of instruction scheduling is across-block analysis. Current methods include *trace scheduling* [11], *superblocking* [16], *hyperblocking* [22], and *wavefront scheduling* [5]. Hyperblocking and wavefront scheduling require special predication hardware, not present in some embedded VLIWs. Superblocking uses a process called *tail duplication* to make multiple copies of many basic blocks. Thus, in situations where code size is even moderately important, superblocking is too costly. Trace scheduling, however, has a more reasonable code size overhead. In trace scheduling, probable execution paths in the program are identified through profiling. Then those basic blocks that are likely to follow one another are assigned to the same trace. The intention is to increase the scheduling flexibility by increasing the shear number of instruction that the within-trace scheduler has available to it. In this way, instructions may end up moving across block boundaries, a process we refer to as *migration*. Yet, in certain cases (as described in Section 6), it may be illegal for an instruction to migrate. Therefore, trace scheduling also provides a migration mechanism to allow only legal moves. This mechanism involves inserting new dependency edges into the trace's data flow graph (DFG), and is transparent to the within-trace scheduler. In addition, once the within-trace scheduler has chosen a schedule, the across-block methods examine the solution to identify which of the instructions, if any, actually do migrate. For these instructions, *compensation code* is inserted into the off-trace path (as described in Section 6). Compensation code will increase code size.

Freudenberger et. al. [12] present an approach to reduce the compensation code created by trace scheduling by avoiding multiple copies of the same compensation code. [12] is an improvement of *dominator-path scheduling* [30]. In [12], specific cases of compensation code are examined, in order to reduce their size. For instance, when an instruction is moved above a multiple-entry point, the

compensation blocks for all of the side entries can be merged into a single compensation block. As another example, compensation copies are avoided along rejoin paths, whenever the copy can be shown to be redundant; a rejoin path is any sequences of control edges where the first edge in the sequence is a side exit from the current trace, and the final edge is side entry back into the trace. In addition to such special-case optimizations, [12] also prevents moving instructions below splits. They note that, though this reduces the available parallelism, it rarely affects performance. We find that, when considering code size as well as run-time, however, this restriction will be too costly, so we *do allow* downward motion, even in our comparison algorithm that is based on [12]. [12] cannot be compared in any meaningful way to our algorithm for two reasons. First, we wish to not only *avoid* the code size cost of trace scheduling, but also to leverage trace scheduling to *reduce* code size, something that is impossible without the within-trace scheduler that we are also proposing. Second, and most importantly, we are not competing against [12]. We have implemented approaches similar to [12] into *both* our method and our comparison algorithm, because [12] represents one of the best existing efforts at reducing the code size increase of across-block instruction scheduling. Therefore, the across-block methods that we propose are built on top of [12]. In this way, our results show the additional benefits of our method beyond this prior work.

Concerning scheduling instructions within a trace, list scheduling, described in Section 3, has proven sufficient for superscalars – they have a re-order buffer to correct bad schedules, and have no constraints on instruction placement. The widespread use of VLIWs in embedded applications has motivated more advanced techniques that consider the resource constraints of the system.

One such technique by Bala et. al. [3] schedules instructions based on finite state automata (FSA) methods in a near-run-time-optimal way. In their method, the resource requirements of each instruction are represented as a bit-vector. If the AND-ing of two bit vectors yields an empty vector, then the two corresponding instructions do not share any resources, and may be scheduled together. Given that a particular set of instructions has been scheduled in a certain cycle, the OR-ing of their bit vectors represents the resources currently used. To construct the FSA, a bit vector of resources used by the set of instructions in the current cycle represents a state, and choosing to schedule an additional instruction on this cycle represents a transition to a new state. Therefore, legal instruction schedules can be identified

as the sentences in a language whose alphabet is given by the instruction set. This makes it easy to ask questions about whether new instructions will fit with others in the same long word. We borrow this approach for keeping track of whether an instruction can be scheduled in the current cycle. Their method goes further in that the scheduling of instructions within a basic block is performed using a straightforward greedy approach: instructions are placed on a cycle until it is filled, and then the next cycle is scheduled. This approach also has application to scheduling across basic blocks, as it is easy to identify whether an instruction can fit into an existing NOP slot. For reasons outlined in the results section, this approach, augmented with the code size reducing techniques in [12], is used as our comparison in evaluating the benefits of our method.

A number of optimal (for run-time) instruction schedulers have been proposed on a variety of systems using ILP formulations [7, 15, 18, 34], genetic algorithms [4], or other similar methods. ILP formulations have the difficulty of not having a bounded compile time. The genetic algorithm of [4] targets a different problem of making a compiler retargetable to different *scalar* architectures. But, *much more importantly*, none of these proposed schedulers considers the problem we address: code size. Since these approaches only target run-time, they perform similarly to [3] in regards to code size.

Our approach of scheduling by branch-and-bound, coupled with heuristics, is also not a new idea. For example, [20] uses branch-and-bound instruction scheduling to reduce register spills in embedded DSP microprocessors. (Our method occurs after register allocation and so is not concerned with spills.) [21] also applies branch-and-bound techniques to instruction scheduling. Their target, however, is to reduce run-time for a very unique VISC architecture. Ultimately, the novelty of our within-trace scheduling technique is the application of branch-and-bound techniques to code size, coupled with aggressive pruning strategies tailored to code size. This is not to say that the code size problem could not be solved with an ILP formulation or a genetic algorithm; but it simply means that we have had good success with a branch-and-bound approach for which optimizing heuristics are easier to derive.

There is also much work on reducing code size through many ways other than instruction scheduling. One popular technique is to compress infrequent portions of code, and then to provide a special software routine to uncompress these regions whenever they are needed, as in [8]. This method is generally applicable to any architecture and yields around a 15% reduction on alpha binaries. In com-

parison, our technique is not limited to infrequent blocks, has little run-time overhead, and does not have the difficulties of self-modifying code. More importantly, since [8] does not involve instruction scheduling, there is no reason why both methods could not be used in conjunction for a cumulative code-size improvement, since our optimized code can be compressed as easily as other code.

In [1], code size is reduced for custom embedded processors by choosing customized templates so as to minimize the NOPs for a given set of applications. This work solves a very different problem of design space exploration for customizing hardware. Our approach could be implemented into part of their system, however, since they use multi-NOPs.

In [6], code size is studied for embedded systems that are programmed by block diagram languages. These languages are based on a model of computation with strong formal properties [9]. This method does not apply to common languages such as C. Our method is performed just prior to code generation, so it is not dependent on the front-end language. (Although our current implementation is for C).

5 Within-trace scheduling

This section presents our within-trace instruction scheduler. Its core is a sophisticated, backtracking instruction scheduler which takes the following approach. It begins with a scheduling algorithm that is provably optimal in that it will find a solution with minimum code size among those with minimum run-time – Section 5.1 details the optimal base algorithm. This algorithm is not feasible, however, since it results in an exponential compile time. Two classes of methods are therefore used to greatly speed up the algorithm. First, Section 5.2 describes aggressive, novel, branch-and-bound techniques that are used to prune portions of the search space while retaining the optimality guarantee. Second, Section 5.3 describes non-optimal heuristics that are used to guide the search towards more promising solutions quickly. Although our backtracker increases the compile-time, this is acceptable since (i) compile-time is less important in embedded systems than in desktops, and (ii) our compile-time overhead is modest.

5.1 Optimal base algorithm

The base algorithm, shown in Figure 2, performs an exhaustive search that returns a provably optimal schedule for each trace. An exhaustive search is impractical, but serves as a good basis for an algorithm that can be modified to be feasible. The search is recursive; at each recursive step, the **for** loop examines

```
SCHEDULE_RECURSIVE(U, PreviousIG, PreviousR)   // Inputs: the set of not-yet-scheduled instructions, the instructions
                                               // scheduled on the last cycle, the instructions ready on the last cycle
  ADVANCE_CLOCK(Clk)                           // Advance the clock
  define Best = MAXINT                         // Initially, no best solution
  if (U = ∅)                                   // See if finished
    return 0
  define R = READY(U)                          // The set of ready-to-schedule instructions
  define NC = NOT_YET_CRITICAL(R,Clk)          // All R which could be delayed
  define C = R − NC                            // All R which must be scheduled this cycle
  for each NC_subset combination of elements of NC
    IG = C + NC_subset                         // this instruction group: all critical + some non-critical
    (Ccost) = FITS_IN_1_LONG_WORD(IG)          // Finds cost of current selection
    if ( not PRUNE(Ccost,U,R-IG,IG,PreviousIG,PreviousR))   // Only explore reasonable choices
      Rcost = SCHEDULE_RECURSIVE(U-IG,IG,R)    // Cost of rest (U-IG)
      cost = Ccost + Rcost
      if (Best > cost)                         // Is this solution the best so far?
        Best = cost
      if (Best = Ucost)                        // Is this a minimum-cost solution?
        return Best
    end if
  end for
  return Best                                  // All possibiltities have been explored
end
```

Figure 2: Optimal base algorithm for instruction scheduling.

all possible *instruction groups*, *IG*, from among the set of ready-to-schedule instructions, *R*. An instruction group is any set of instructions that can execute in parallel. IG is often an unordered set, but some VLIWs allow ordering restrictions within a group, for example to support memory disambiguation.

In Figure 2, every IG must contain all critical instructions[1], *C*, as well some subset, NC_{subset}, of the non-critical-but-ready instructions. For each chosen *IG* that can schedule within one long word, a recursive call finds the best schedule of the remaining code, and the scheduling clock advances. If an IG contains fewer instructions than the width of the VLIW machine, the remaining slots are padded with NOPs. In its present formulation, Figure 2 is for fixed width VLIWs, but the modifications are slight to accommodate systems with variable-length long words, such as multi-NOP or EPIC machines.

To identify the critical instructions, however, we must first know how many cycles are needed to schedule the entire trace. Although this number cannot be easily calculated [4], we may estimate it as the maximum of two lower bounds. One lower bound is the trace's DFG height. Another is found from resource usage. Thereafter the algorithm in Figure 2 uses this estimate. If it then finishes without finding a solution, all instruction deadlines are increased by 1, and the algorithm is rerun. For most traces, the original estimate is correct. In this way, only schedules of minimum run-time are considered.

[1] An unscheduled instruction is said to be critical (or to have met its deadline) if, intuitively, delaying it further will mean that a minimum latency solution cannot be obtained on this path. The minimum latency is the latency of the longest latency path in the dataflow graph. Mathematically, the condition for an instruction to be critical is when *current schedule cycle + longest path latency from this instruction to the bottom of the DFG = minimum # cycles required to schedule the trace.*

Cost metric The goal of our search technique is to find the schedule with the smallest code size from among those having the minimum run-time. Thus the first constraint for scheduling is run-time. Because instructions must schedule by their deadlines, *Figure 2 always finds a minimum run-time solution.* In this figure, the code size cost is measured in NOPs, because, with the number of useful instructions being fixed, the number of NOPs relates directly to the code size.

Impact of register allocation Instruction scheduling and register allocation are interdependent. The register allocator maps variables onto machine registers. Since a function may contain more variables than there are physical registers, the allocator must reuse registers. To reduce the cost of register spills, the allocator attempts to assign variables in such a way as to minimize the number of registers used by a procedure. If instruction scheduling is performed prior to register allocation, then the live ranges of all variables become fixed, and the allocator will not be able to reduce the number of registers as much as it otherwise could. On the other hand, if register allocation is performed first, then different variables will be mapped to the same physical register, thereby introducing *anti* and *output* dependencies into the DFG. And adding edges into the DFG will restrict the instruction scheduler, worsening its result.

We have chosen to implement our scheduler *after* register allocation. This avoids four problems that arise when scheduling before allocation. First, our algorithm would need a new heuristic for reducing register pressure in the schedule. Second, our schedule could cause new register spills. Third, such register spills generally require the allocator to insert new "spill" instructions that might not fit into the existing schedule. Fourth, the implementation would be harder, requiring modifications to the register allocator. But scheduling instructions on the final pass avoids all these problems, and allows us to ignore register allocation, by treating anti and output dependencies just like true dependencies. We do note, however, that by moving our pass prior to register allocation, we could remove edges from the DFG, thereby improving the scheduling flexibility, which is likely to further *improve* the results.

5.2 Optimal pruning

Since the run-time of our exhaustive search is infeasible, we use branch-and-bound techniques. Branch-and-bound techniques can drastically reduce the execution time while retaining the optimality guarantee, by *pruning* (*i.e.,* skipping) parts of the search space that are known to not contain an optimal

```
PRUNE(Ccost,U, L, IG, PreviousIG, NewR)  // Inputs: current cost, the set of unscheduled instructions, the set of ready-but-not-chosen
// instructions, the chosen instructions, the instructions chosen on previous cycle, and the set of instructions that just became ready this cycle

// COST_PRUNING: rejects solutions which are more expensive than the current best
   if (PartialCostOfAlreadyScheduled + Ccost + LOW_BOUND(U)≥ BestCompleteSolutionSoFar)
         return 1

// NOT_FILLED: looks for an unscheduled-but-ready instruction that can schedule with IG
   if (∃ an instruction, i ∈ L, such that SCHEDULABLE(IG + i))
         return 1

// NOT_NEW: looks for cases where the last cycle was empty, and none of the scheduled instructions are new
   if ((PreviousIG = ∅) and (IG ≠ ∅) and (∄ an instruction, i ∈ IG such that i ∈ NewR))
         return 1

// SAME_ECLASS: checks whether the scheduled instructions of IG belong the same e-classes as a previously tried case
   if (∃ an e-class, E ∈ ECLASSES_USED(IG), such that E's first unscheduled instruction ∉ IG)
         return 1

// STRICTER_ECLASSES_WORKED: checks whether the instructions of IG match to stricter e-classes in a previous schedule
   if (∃ previous schedule, PS, such that, for ∀ E ∈ ECLASSES_USED(IG) , E maps to a stricter or same e-class in ECLASSES_USED(PS))
         return 1

   return 0
end
```

Figure 3: Pruning techniques that run in linear time

solution, or if the remaining search space is guaranteed to have an optimal solution. Figure 2 contains a call to the *PRUNE()* procedure. If *PRUNE()* returns TRUE then the current path is abandoned by skipping to the next iteration of the **for** loop; otherwise it is explored further by making a recursive call.

Figure 3 shows our five pruning strategies. These are general techniques, applicable to any of the VLIWs that our approach targets; other machine-specific pruning strategies can also be imagined.

This paragraph describes the first test from Figure 3, *COST_PRUNING*. COST_PRUNING is applied to both the run-time cost *and* the code-size cost. In either case, if the lowest possible cost of the current schedule is greater than or equal to the best solution found so far, then this path is pruned. The lowest possible cost for the current schedule is computed as the sum of the cost of scheduling up to the current point plus a lower bound on the cost of scheduling the remaining instructions. For run-time, this lower bound is derived from the DFG height of the not-yet-scheduled instructions, and from these remaining instruction's resource requirements. For code size, the lower bound for the remaining cost is computed as the largest of *three separate lower bounds*. The first is based on resources: if a certain machine allows two floating point instructions per cycle, and if 7 floating point instructions remain to be scheduled, then at least 4 long words are needed. The second bound is based on the issue width: in a 6-wide VLIW, scheduling 13 instructions requires at least 3 long words. The third is based on those instruction that lie on any longest path in the remaining DFG. These instructions are called FIXED_CYCLE instructions, because, in a schedule of minimum run-time, the cycle on which they

become ready is also their deadline. The number of such cycles is the third lower bound on the code size. We refer to the largest of these three lower bounds as *#Long_Words_LB*. It follows that a lower bound on the NOP cost is: *NOP_LB = (VLIW_width * #Long_Words_LB) - #Unscheduled_Instructions*.

The second method of Figure 3 is NOT_FILLED pruning. To see its motivation, consider a yet-to-be-scheduled instruction, *A*, that is not in the currently-chosen-to-be-scheduled set of instructions, IG (*i.e.*, $A \in R,\ \ A \notin IG$). If scheduling IG leaves unfilled slots (NOPs in its long word), then it never sacrifices optimality to schedule A in one of those slots instead of leaving it empty. Thus the schedule without A can be pruned. A similar argument can be constructed for multi-NOP or EPIC architectures.

The third pruning method is NOT_NEW. Its intuition is that the SCHEDULE_RECURSIVE() procedure of Figure 2 is called every cycle, and different cycle-by-cycle search sequences in the algorithm can correspond to the same final code. Figure 4 shows an example of how this can happen in a 2-wide VLIW. For the DFG shown in Figure 4(a), Instruction #2 may schedule on any cycle between 2 and 10. The resulting code is the same, however, and is shown in Figure 4(b). To avoid this redundancy, we place a restriction on the current instruction group, IG_{cur}, based on the choice of instruction group on the previous cycle, IG_{prev}. The restriction is that if IG_{prev} is empty (*i.e.*, a stall cycle), then IG_{cur} must either be empty or contain at least one instruction that was not available on the previous cycle. The reason for this is that, if all of the instructions in IG_{cur} had been available on the last cycle, then swapping IG_{cur} and IG_{prev} would also be a legal schedule. Preventing such swappable situations solves the problem of Figure 4, while maintaining optimality. (For understanding Figure 4, we remind the reader that an instruction may have two different outgoing latencies due to an anti-dependence.)

The fourth pruning method is SAME_ECLASS. It targets instructions that are equivalent from the scheduler's viewpoint. Such instructions must use exactly the same resources, and contain exactly the same outgoing DFG edges, with the same weights. Thus if one of these instructions was already tried as a member of IG then there is no need to try the other separately. For our test machine, we measured an average of 1.3 instructions per e-class. While not a large number, the exponential nature of the search space makes this optimization worthwhile.

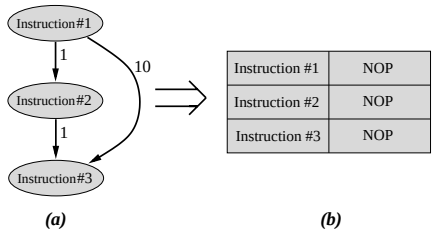


Figure 4: Example of how scheduling an instruction on different cycles yields the same final code.

The fifth pruning method is STRICTER_ECLASSES_WORKED. We define an equivalence class, E1, to be *stricter* than another, E2, under two conditions. First, instructions from E2 must use no resources that are not used by E1. This happens when E1 and E2 are the same type, or when E1 is a subtype[2] of E2. Second, the instructions of E2 must have no out-going dependency edges that are not also in E1, and none of the outgoing edge latencies of E2 may be larger than the corresponding latencies of E1. Under these conditions, E1 is strictly harder to schedule than E2. Therefore if E1 succeeds in scheduling then E2 is not attempted. In our benchmarks, each e-class tends to have one weaker e-class.

Where as the five methods in Figure 3 identify *bad* choices to skip, it is also possible to identify *good* solutions and quit early (without considering the remaining combinations of C). If a given solution is provably optimal (*i.e.*, its cost equals the lower bound) then there is no need to search for a better solution. This optimization is indicated in Figure 2 by means of the early return from inside the loop.

5.3 Non-optimal heuristics

Although branch-and-bound greatly reduces the search space, the compile time of some large, complex traces is still unmanageable. In such cases, non-optimal heuristics are needed. Heuristics reduce the compile time by searching only a portion of the solution space, without guaranteeing optimality. We use such heuristics only for larger traces that have already taken a long time to compile. A desirable consequence is that smaller traces, and larger ones that finish quickly, will be scheduled optimally.

We investigated many heuristics, and identified three that are effective. *First*, traces longer than an experimentally determined size (currently 35 instructions) are split into *trace chunks* of that size. Smaller traces will have a single trace chunk. *Second*, extra edges are added between FIXED_CYCLE instructions if they exceed the resource constraints. Recall that a FIXED_CYCLE instruction can only become ready on the same cycle as its deadline; so, the exact clock when it will schedule is known in advance. Therefore, if several instructions share the same FIXED_CYCLE, we may detect in advance if they can all fit into one long word. If not, one might simply increment all deadlines before scheduling, by the same reasoning used in Section 5.1. Such an approach, however, was found to waste time considering many unwise choices. By instead placing an edge of latency 1 between conflicting FIXED_CYCLE instructions, the scheduling bottleneck is directly tackled, while still often finding an

[2]Some VLIWs define subtypes, which are instructions with additional restrictions beyond those of their parent type. For instance an integer subtype might only be allowed to schedule to the first IALU

optimal solution. *Third*, whenever the scheduler runs too long, it will time out, taking the best solution found so far. The time-out value for a given chunk is based on its complexity, using an empirically-determined function described in the results. For now, it is enough to know that simple traces are *guaranteed* to finish within 5 ms, and that, regardless of complexity, every trace chunk is guaranteed to finish within 1000 ms. This provides worst-case-compile-time guarantees for a given program. For our benchmarks, this worst-case bound is, on average, a factor of 6 of the original compile time.

6 Across-block scheduling

In both our algorithm and the comparison algorithm, we employ trace scheduling [11] with the code size improvements of [12]. The new feature of our across-block analysis is how we constrain the movement of instructions between basic blocks, called *migration*. We propose preventing migration when it is likely to increase code size. In this way, we avoid the code-size costs of trace scheduling while at the same time enjoying the code-size benefits that it offers. As with other across-block schedulers, once a trace has been constrained, it is sent to the within-trace algorithm for scheduling. The rest of this section describes compensation code and then describes a compile-time analysis to detect if compensation will increase code size. In fact, allowing compensation code can sometimes *reduce* code size.

Examples of compensation code Compensation code [11] refers to the duplication of an instruction when it migrates to a new block. Figure 5(a) illustrates the use of compensation. In this figure, BB #1 and #2 are part of the same trace. The scheduler has decided to migrate *Instruction A* from BB #1 into BB #2. While the execution of BB #1 will probably be followed by BB #2 (*i.e.*, they are in the same trace), it is also possible that the side exit to BB #4 will be taken. Then, unless a copy of *Instruction A* is placed along this path, the program behavior will be incorrect. Nor can *Instruction A* simply go into BB #4. In that case, the execution path from BB #3 to BB #4 would be incorrect. Figure 5(b) illustrates the solution of traditional trace scheduling, which duplicates instruction A to a new basic block, BB #5.

Similarly, Figure 5(d) describes the situation of an instruction moving up above a side entry. *Instruction A* cannot be copied into BB #3, because BB #3 may proceed to BB #4. The solution is to again create a new basic block for the compensation code, as shown in Figure 5(e). Figures 5(b) and (e) show only two of the four possible cases. However, existing across-block schedulers do not attempt to migrate instructions above a side exit or below a side entry, because it would make the program unsafe.

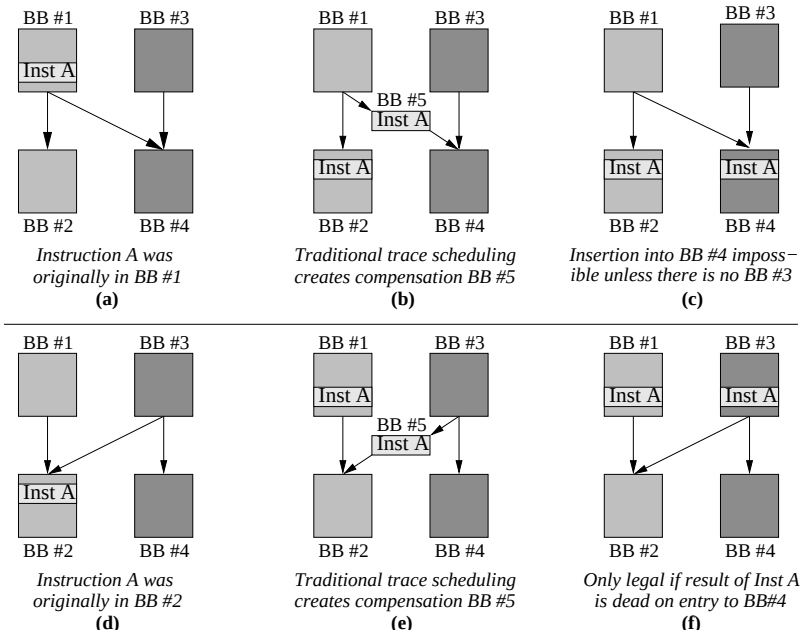


Figure 5: Scenarios for inserting compensation code. BB #1 and BB #2 are part of the trace. BB #3 and BB#4 are outside of the trace. In (a)-(c), *Instruction A* was originally in BB #1, but has been scheduled into BB #2. Compensation code is needed for the side exit to BB #4. In (d)-(f), *Instruction A* was originally in BB #2, but has been scheduled into BB #1. Compensation code is needed for the side entry from BB #3. (b) and (e) illustrate the solution of traditional trace scheduling. (c) and (f) illustrate our solution, which is not always feasible.

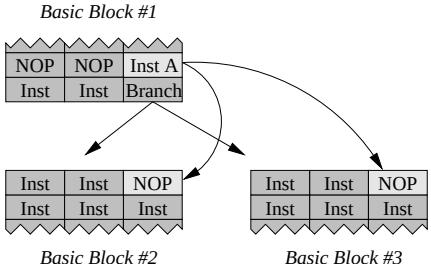


Figure 6: An example of how code size can be reduced even though an instruction is duplicated. Basic Blocks #1 and #2 are part of one trace, and Basic Block #3 is a side exit. If *Instruction A* is moved into Basic Block #2, it will fit into a NOP slot. The compensation copy placed into Basic Block #3 also fills a NOP slot. By moving *Instruction A* out of Basic Block #1, one of its long words becomes empty and can be removed.

Potential increase in code size The creation of new basic blocks for compensation code (BB #5 in Figures 5(b) and 5(e)) increases code size. This increase is particularly costly since only one instruction might migrate, as in BB #5, yet an entire VLIW long word would still be needed on some architectures.

How code size increase can be avoided Our method is to detect, through compiler analysis, those instances of code motion that do not increase code size, and to disallow all other instances of code motion. The rest of this section discusses how to detect when code size will increase.

We avoid the code size increase of compensation in the following two cases. First, a duplicate instruction is not needed in the off-trace path if its destination register is dead along that path. For example, in Figure 5(b), *Instruction A* can indeed be copied into both BB #2 and BB #4, if the destination register of *Instruction A* is dead upon entry into BB #4. Second, we can avoid creating extra long words for compensation if two conditions hold. (i) If the off-trace duplicate instruction can move

into an existing basic block, rather than creating a new block. For example, in Figure 5(b) instead of creating the new basic block BB #5, the duplicate copy of *Instruction A* can go directly into BB #4 *if* there is no BB #3 (BB #4 has only one parent). (ii) If, in addition, the off-trace duplicate replaces an existing NOP in the side block. For example, *Instruction A* in BB #4 does not increase code size if it fits into an existing NOP position within BB #4.

Paradoxically, increasing the number of instructions, through compensation code, can have the effect of *reducing* the code size. Figure 6 describes this situation. *Instruction A* is moved from block #1 to blocks #2 and #3, fits in existing NOPs in both those blocks, and the entire long-word in block #1 is eliminated, reducing the net code size by one long word. Our algorithm needs no special case for handling such scenarios. The benefit is naturally obtained as a sub-case of the second case above, where our scheduler allows the instruction to migrate since it does not increase code size. The fact that the code size is reduced is a bonus that is only later discovered by the within-trace scheduler.

Our approach would perform even better if predication [24] were present, because the legality restrictions described in 5(c) and 5(f) could be removed. Predication is becoming more common in embedded VLIWs [29, 31], however it is not universal. Our results are measured without predication, but would only improve if it were present.

Implementation Here we specify how the compiler discovers the two cases, listed above, of when code motion does not increase code size. The first case is when a duplicated instruction is not needed in the off-trace path because its destination register is dead along that path. This case is easily discovered by the compiler through liveness analysis, which is a widely used form of dataflow analysis. The second case is when a duplicated off-trace instruction (i) can be placed into an existing basic block and (ii) in addition, replaces an existing NOP in that block. Detecting condition (i) only requires the control flow graph. However, detecting when (ii) is possible is hard since, to know whether an instruction can replace an existing NOP, we need to have already finished scheduling the side block!

This problem of identifying when compensation instructions fit in NOPs is solved by *preliminarily* scheduling each basic block separately. This schedule provides an initial estimate to the across-block analyzer that must identify when compensation code fits into existing NOPs. After our across-block analysis is performed, within-trace scheduling begins. One complication at this point is that multiple

compensation instructions may contend for a small number of NOP slots in the side block. Therefore, the scheduler must ensure that there are enough NOP slots for all instructions that it chooses to migrate. Finally, the preliminary schedules are discarded.

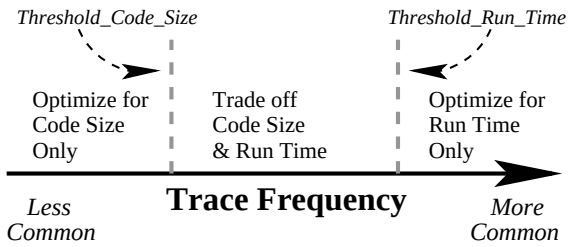


Figure 7: Scheduling strategies along the trace frequency spectrum. Traces are sorted by frequency. Those whose frequency is below an experimentally determined threshold, *Threshold_Code_Size*, are scheduled for code size only, and for those whose frequency exceeds *Threshold_Run_Time* only run-time is targeted.

7 Considering extremes

Up until now, our approach has attempted to minimize both run-time and code size. In Section 5, we find the smallest schedule among those with minimum run-time. In Section 6, we use compensation code to reduce the run-time, but only when it does not increase code size. But in this section we observe that we may sometimes do better by removing one of these constraints, either on code size or run-time. For the most frequent traces, it is reasonable to optimize for run-time only, even at the cost of code size. In a complementary way, the most *in*frequent traces should be optimized for code size only, even at the cost of run-time. Figure 7 pictorially depicts these observations. Such a tradeoff between two objectives is new since it is not needed for existing methods that only optimize for run-time in all cases.

Frequent traces For the most frequent traces, run-time becomes more important than code size. To adapt our methods for a goal of run-time only, the within-trace methods of Sections 5 require no modification, as they already search for a minimum run-time solution. The across-block analysis of Section 6 could negatively affect the run-time, however, because it restricts code motion. Therefore, for frequent traces, we do not constrain migration with the methods of Section 6. The frequent traces are defined as the set of the most frequent traces in the program whose combined run-time is a threshold percentage of the program's total run-time. This threshold is called $Threshold\_Run\_Time$ and is determined one-time in the results section by repeating our method with different thresholds and choosing the best.

Infrequent traces For the least frequent traces, code size is the over-riding concern. This requires a modification to the within-trace methods of Sections 5, but not to the across-block restrictions in Section 6. Our within-trace methods are modified to optimize for code size alone by simply setting all

data dependence latencies to a value of 1. This is because, when the dependence latencies are all 1, the minimum run-time solution *is the same* as the minimum code size solution, since both are directly proportional to the number of long words. The resulting solution is still correct for the original latencies by inserting stalls where appropriate, and the code size is optimal since the minimum code size solution does not depend on the instruction latencies. The least frequent traces are defined as the set of the most least frequent traces in the program whose combined run-time is a threshold percentage of the total run-time of the program. This threshold is called $Threshold\_Code\_Size$ and is determined one-time in the results section by repeating our method with different thresholds and choosing the best.

8 Results

Evaluation environment and benchmarks Because of compiler source code availability and access to a physical processor, we chose our evaluation platform to be a hypothetical 6-wide embedded VLIW whose instruction set is identical to the Itanium [2, 28], an Intel desktop VLIW. A 3-wide VLIW is also evaluated. The variable-width long words feature of the Itanium is not allowed, however, so as to represent a more typical VLIW processor for embedded systems. The speculative load and hyperblocking features are also not permitted, on similar grounds. A 6-wide VLIW is not excessive for embedded systems; for example, the TI C6000 series uses a width of 8. Although we model a fixed-width VLIW, the compiled code is still compatible with the IA-64 ISA, and so it is evaluated on a real Itanium.

An IA-64 instruction set was chosen since the most sophisticated open-source VLIW compiler we could find was for the Itanium instruction set. Using a sophisticated VLIW compiler has two advantages: first, it reduces the implementation effort since much functionality already exists, and second, by using a mature VLIW compiler, we can have confidence that our scheme improves performance versus the best available schemes. Our algorithm is implemented by modifying the PRO64 (v 0.11) research compiler for Itanium [27], run with full optimization.

Some of the important parameters of our test machine are as follows. It has two memory units, two integer units, two floating point units, and three branch units. Each of these unit types is asymmetric. For instance, some M-type instructions are only executable on one of the memory units, while others are executable on both. There is also a supertype instruction class, A, that can execute on either an I or M unit, but which has different latencies depending on the unit that it executes on. It also may

Name	Description	Type	# lines of code	# Basic Blocks*	# Useful Operations*
adpcm	Converts 16 bit PCM and 4bit ADPCM	telecom	943	74	467
fft	Discrete Fast Fourier Transforms	telecom	331	123	750
basicmath	math functions, like sqrt and sine	automobl	315	117	1038
bitcount	counts the number of bits in data	automobl	644	87	660
dijkstra	Dijkstra's algorithm for shortest paths	network	177	66	411
jpeg	Compress and decompress jpeg images	consumer	51025	11767	94979
blowfish	The encryption algorithm	security	3136	131	1992
rjindael	The AES encryption algorithm	security	1138	187	3913
sha	The Secure Hash Algorithm	security	242	82	901
stringsrch.	Searches for strings in text	office	3037	187	1448

***** *found by examining the output of* PRO64

Table 1: Benchmarks. All are in C and from the MiBench suite.

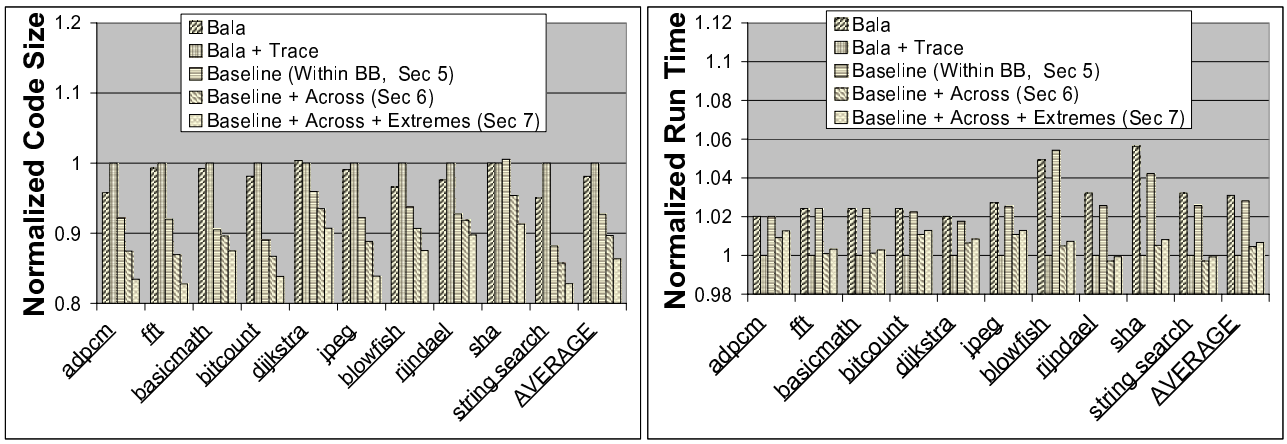
place ordering restrictions upon the instructions that are run in parallel. All of these features increase the difficulty of the compiler's task, and thus challenge our method more. Our approach is equally applicable to less constrained systems. Experiments are run on ten benchmarks from the MiBench suite of embedded applications [14]. All are written in C. The benchmarks are described in Table 1.

Comparison algorithm Our algorithm is compared to an implementation of the FSA-based near-run-time-optimal across-block scheduler in [3] by Bala et. al., which is augmented by the techniques in [12] by Freudenberger et. al. so as to reduce the size of compensation code. This choice allows us to compare against one of the best across-block schedulers that aims for run-time reduction ([3]), combined with one of the only techniques for reducing code size in such an approach([12]). There are better-for-run-time schedulers than [3], but most of them, such as superblocking [16], increase code size dramatically more than [3], and so a comparison against them would make our code size reduction look unrealistically good. Further we will show later, in Figure 8, that when applied to single basic blocks, [3] yields nearly the same run-time as our provably-optimal-in-run-time within-trace scheduler, providing strong verification that [3] is a near-optimal-in-run-time scheduler. The best compiler technique for explicitly reducing code size that we are aware of for our targeted subset of VLIWs is [12], and so we compare against an implementation of [3] augmented with [12]. See Section 4 for a fuller discussion of the related work. In the rest of this section, 'Bala' is shorthand for our comparison algorithm of Bala et. al. [3] augmented with Freudenberger et. al. [12].

Experiments Figure 8 shows the code size, run-time, and compile time for five methods: (i) Bala applied within basic blocks, (ii) Bala with traces (normalized to 1.0), (iii) our within-trace scheduler applied separately to individual basic blocks, (iv) our within-trace scheduler with across-block analysis, and finally (v) our complete scheduler including the changed optimization criteria at the extremes. Thus, the second bar represents current methods, and the fifth bar represents our method. By comparing the two, we see that our method reduces the code size by 15.8% compared to the best previous method, while staying within 0.82% of its run-time. In each figure, the average is shown as the right-most set of bars for convenience. In Figure 8 (c), we see the compile time for each benchmark, graphed against the theoretical worst-case compile time. The worst case compile-time is found by assuming that the branch-and-bound scheduler never finishes before its timeout value. The derivation of the timeout value will be described in Figure 10. In reality, the timeout is rarely used; but this upper-bound value provides compile-time guarantees, ensuring that our methods never require exponential run-time. From Figure 8 (c), we find that most programs are *guaranteed* to finish within a factor of 6 times their original compile time, but in reality finish with just a factor of 2 increase. Such a compile-time increase is generally acceptable in embedded systems, because compilation occurs in the factory.

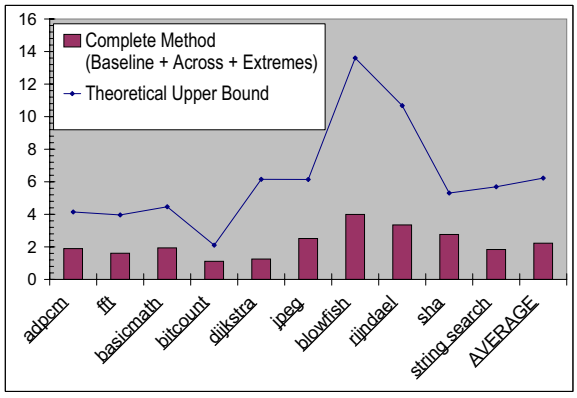
Figure 8 also reveals the incremental contributions of our different technologies to the total code-size improvement. First, by comparing the first bar versus the third one, we see that our within-trace technologies improve code size by 6.4% while maintaining the run-time of current within-trace schedulers. In fact the run-time is slightly faster, because the Bala algorithm is greedy. But the closeness of the run-times is strong evidence that the Bala algorithm is doing a good job of finding a minimum run-time solution (albeit with a worse code size). Second, by comparing the third bar versus the fourth, we see that our across-block technologies contribute an extra 5.3% code-size improvement versus our within-trace methods. Third, by comparing the fourth bar versus the fifth, we see that changing the optimization criteria at the extremes improves code size by a further 2.4%.

Figure 9 breaks down the contributions of our various pruning strategies to code size, run-time, and compile time. In Figure 9(a), (b) and (c) the right-most bar for each benchmark represents our final algorithm. Figure 9(a) indicates a 15.8% savings in code-size, in agreement with the rightmost bar of Figure 8. Figure 9(b) reveals that the pruning techniques have little effect on run-time. This is because



(a)

(b)



(c)

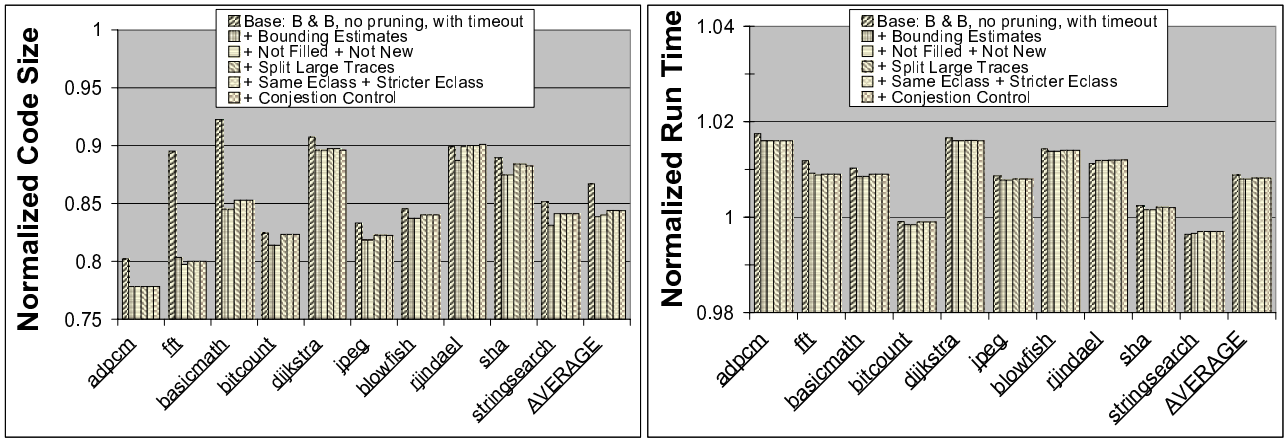
Figure 8: (a) Improvement in code size, (b) run-time achieved, and (c) compile-time costs for the methods of Sections 5, 6 and 7, normalized to Bala with traces = 1.0.

minimum-run-time solutions are easier to find than solutions with both minimum run-time and code size. In fact, before performing branch-and-bound, an initial bound for the best-schedule-found-so-far is obtained through a greedy search similar to Bala, and Figure 8(b) has already shown that Bala produces a comparable run-time to our approach. Therefore there is little chance for improvement in run-time. Instead, the effect of pruning strategies is to reduce the compile time, and as a consequence, to allow a larger portion of the design space to be searched before timing out – perhaps leading to the discovery of a more-compact schedule. Finally, Figure 9(c) reveals that the compile-time overhead of our algorithm is a factor of 2.2, with no case being worse than a factor of 4. This overhead is quite reasonable for embedded systems, where code is often compiled only in the factory.

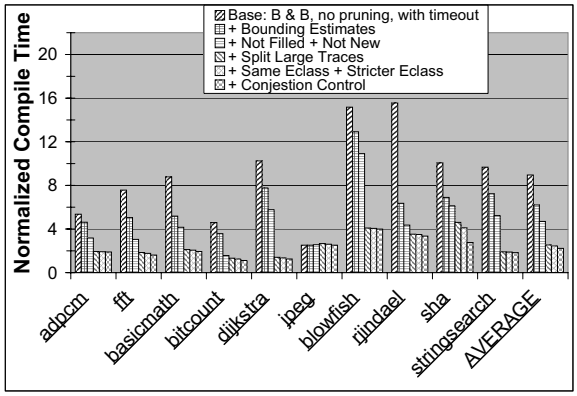
Let us now consider the effect of the pruning strategies in Figure 9. The first bar represents our algorithm without any pruning, except for a timeout. The impact of the timeout heuristic cannot be

measured, since without it a few large traces did not finish even in days, although most traces finished within the timeout. Only a small number of traces timed out after 1 second, and increasing the timeout did not significantly reduce the number of timeouts or improve the code size. This base approach demonstrates a 15.3 % code-size improvement and a factor of 8.6 increase in compile time versus the compiler without our techniques. The second bar represents the addition of lower bound pruning to the base algorithm. Where as the first bar does not consider the cost of the as-yet-unscheduled instructions, the algorithm of the second bar considers lower bounds on the remaining cost, thus providing a tighter bound for pruning. This pruning technique demonstrates a further 2.6% decrease in code size, and 30.6% reduction in compile time. The third bar indicates the pruning technique whereby certain redundant or provably inferior schedules are skipped. This reduces the code size by a further 0.19% and the compile time by 24.4 %. The fourth bar reveals that splitting large traces into more manageable *trace chunks* (of 35 instructions each) *increases* the code size by 0.46% over the third bar, but reduces the compile time by 45.8%. Splitting large traces is a non-optimal heuristic, so it is possible to increase the code size. Yet, since this heuristic has such a positive effect on compile time, a designer may choose to compile with this flag, as we have. Next, the fifth bar considers the effect of adding equivalence class analysis to remove redundancies among possible instruction groups. This technique had no measured effect on code size, but decreased compile time by 3.6%. Lastly, the sixth bar represents our final algorithm, which uses all of the pruning techniques. This final bar shows the effect of congestion control, where edges are inserted between constrained, FIXED_CYCLE instructions. This pruning technique reduces code size by only 0.02%, but reduces the compile time by an additional 9.18%.

Figure 10(a) displays the actual compile time of every trace chunk from every benchmark. Each of the 8,390 data points represents one trace chunk: the y-value being the chunk's measured compile time and the x-value being computed as a simple function of the chunk's properties. We have developed this function so as to estimate a chunk's compile time before scheduling it. This function is the product of two terms. The first term is the number of instructions, $N_{instructions}$, plus the number of instructions not on a fixed cycle, $N_{flexible}$. The intuition here is that instructions with scheduling flexibility will increase the search space, and so they affect the compile time more than fixed instructions. The second term also attempts to measure the size of the search space: it is the natural log of the product of all slacks

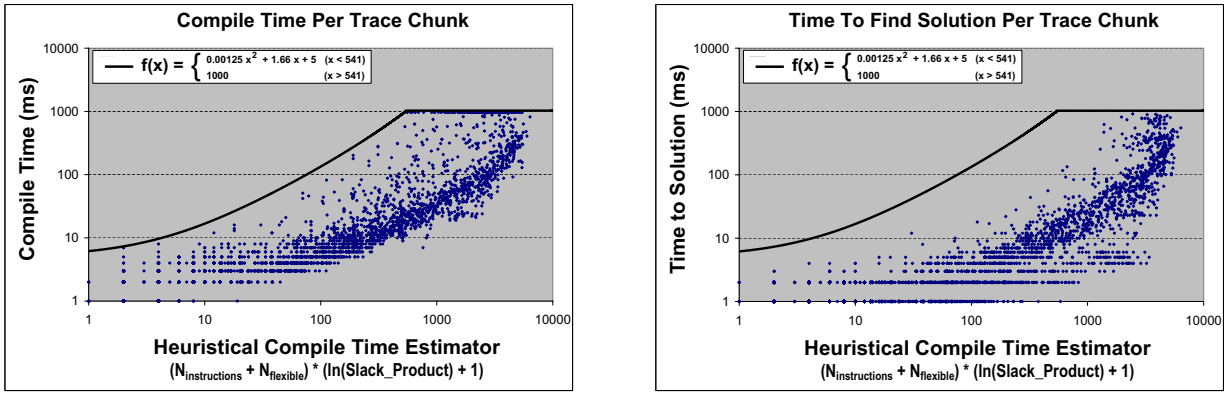


(a) (b)



(c)

Figure 9: The effect of our pruning strategies upon (a) code size, (b) run-time, and (c) compile time, normalized to Bala with traces (Bala does not use pruning). Our proposed algorithm is represented by the left-most bar.



(a) (b)

Figure 10: Determining the proper time-out function. In (a), the actual measured compile time for each trace chunk is plotted versus a simple heuristic estimator of compile time. In (b) the time when the final solution was reached is given. In both figures, our proposed function is also plotted, chosen by fitting a parabola two the upper 2% of data points, while requiring the parabola's y-intercept to be 5 ms.

in the DFG, plus one. In Figure 10(b), this same estimator function is used for the x-value of the data points, but the y-value is the time at which the final solution was found. Therefore, the data points in Figure 10(b) are lower than in (a), because a branch-and-bound algorithm *may* search for a long time without finding a better solution than one visited early in the search.

Figure 10 also illustrates our timeout function. By fitting a parabola through the top 2% of data points in Figure 10(a), while requiring the y-intercept to be 5 ms, we arrive at the timeout function, f(x), plotted in Figure 10. The y-intercept restriction is needed so as to ensure that the smallest chunks are given sufficient time to compile, which turns out to be 5 ms. In this figure, we also see that the timeout function is clipped at 1000 ms. Examining Figure 10(a), we notice that almost all trace chunks finish compilation without timing out (95%). In Figure 10(b), we further observe that an even larger number of chunks will have found their final solution before timing out. As a result, the time-out function has a marginal impact on the quality of our results, but a major impact on the quality of our compile-time guarantees.

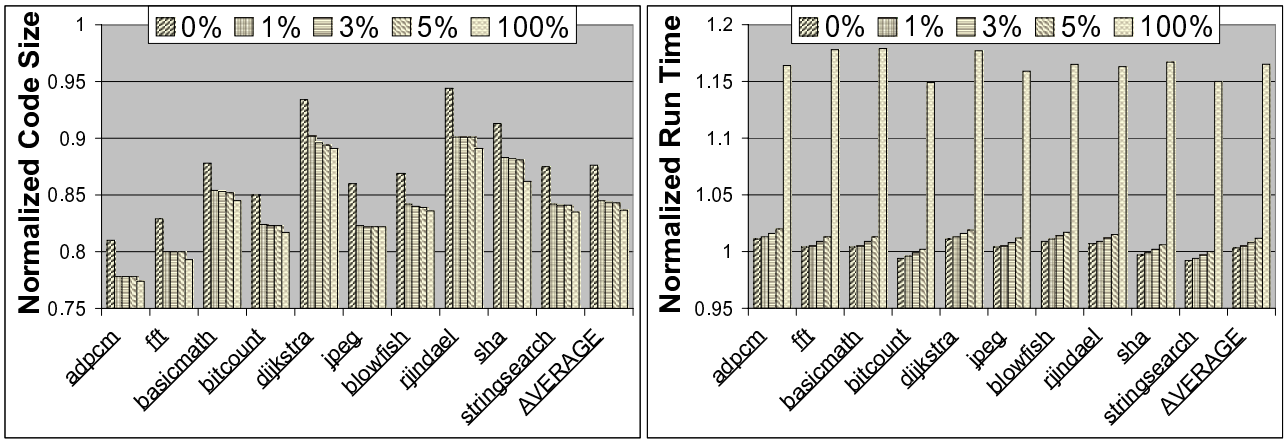
Figures 11 and Figure 12 illustrate how the thresholds of Section 7 are chosen. Some explanation of the meaning of these thresholds is warranted. If Threshold_Code_Size = X this indicates that only code size matters for the set of least frequent traces whose combined execution time is X% of the total execution time of the program. Similarly, if Threshold_Run_Time = Y this indicates that only run-time matters for the set of most frequent traces whose combined execution time is (100 - Y)% of the total execution time of the program. The sense of this definition is reversed so that the two thresholds can be understood together, with common units, as in Figure 7. Thus for both thresholds, a higher value corresponds to a higher preference of code size versus run-time. When no thresholds are used (as in Section 6, and corresponding to the fourth bar of Figure 8), the effective values of these thresholds could be considered to be: Threshold_Code_Size = 0% and Threshold_Run_Time = 100%.

Figure 11 illustrates our method for determining the optimal value for the Threshold_Code_Size parameter. In this figure, the other parameter – Threshold_Run_Time – is set to its optimal value (15%). The experiments in Figures 11 and Figure 12 both require the solution to the optimal threshold in the other figure. To solve this 'chicken and egg' problem, we iteratively repeated the experiment in each figure until both thresholds converged. We see from Figure 11 that choosing 3% for Threshold_Code_Size

achieves most of the code-size improvement of limiting compensation code, while at the same time achieving most of the run-time benefit of trace scheduling. This is because most of the execution time is spent in a small number of traces.

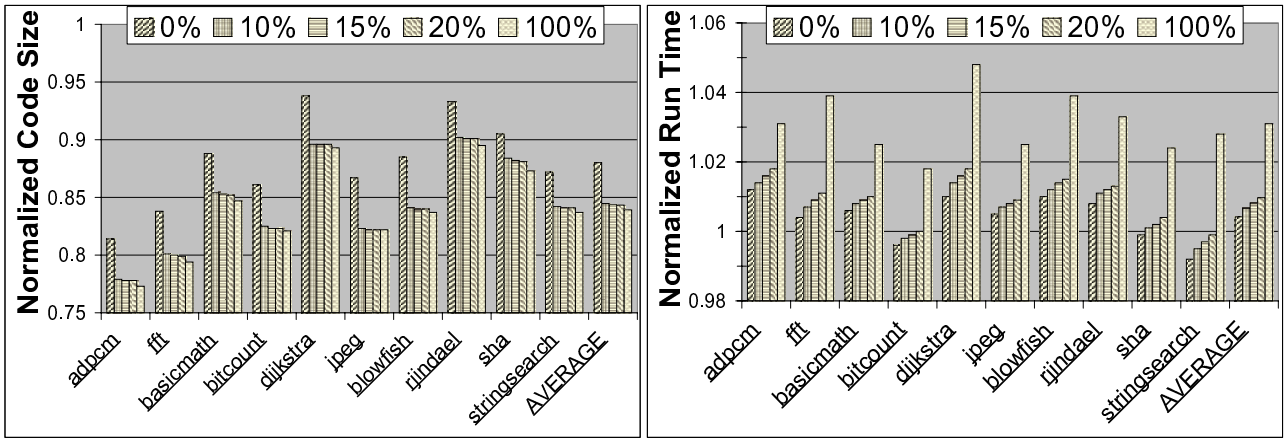
Similarly, Figure 12 depicts the effect of the Threshold_Run_Time parameter, with Threshold_Code_Size set to its optimal value (3%). This figure shows that choosing a value of 15% for Threshold_Run_Time maintains most of the code size improvement of the code-size-only schedule, without a serious impact on run-time.

It is interesting to see how VLIW width affects our results. Since IA-64 packs 3 instructions into each bundle, our implementation can be modified to compile for a 3-wide VLIW, instead of a 6-wide.



(a) (b)

Figure 11: The results of varying the Threshold_Code_Size parameter, upon (a) code size and (b) run-time, normalized to Bala with traces. (Threshold_Run_Time is held constant).

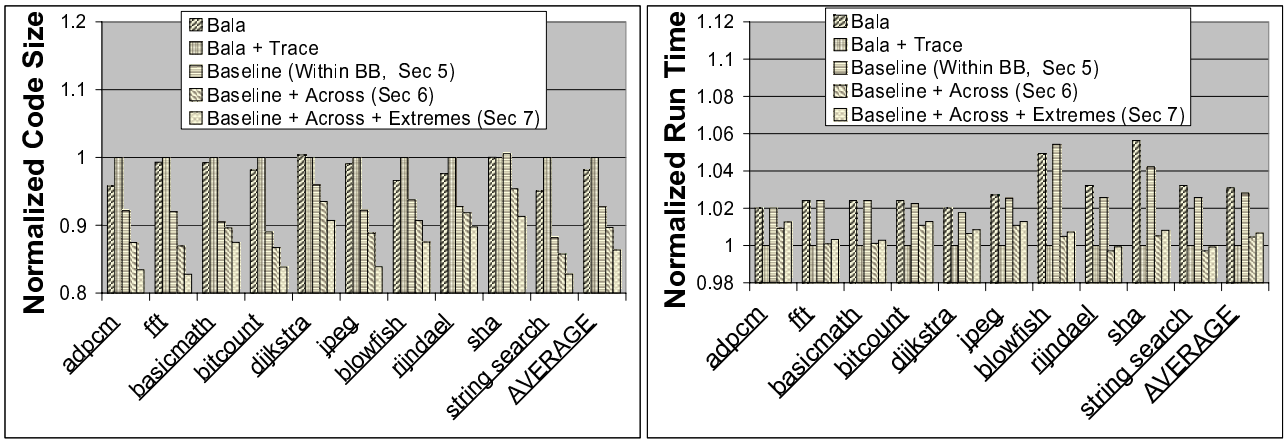


(a) (b)

Figure 12: The results of varying the Threshold_Run_Time parameter, upon (a) code size and (b) run-time, normalized to Bala with traces. (Threshold_Code_Size is held constant).

Figure 13 describes our 3-wide results. Comparing Figures 13 and 8, we make two observations. First, the average code size reduction for a 3-wide VLIW is 13.6%, which is 86% of the improvement found in the 6-wide VLIW. Hence, the code-size improvement of our method scales downwards reasonably well. This information also suggests that our technique may have a somewhat larger effect on even wider VLIWs. Second, the basic trends in the figures for the two widths are similar.

It is also interesting to see the relative proportion of trace lengths in our benchmarks. Figure 14 divides the traces according to their number of useful instructions, and gives the percentage of traces in each range. We see that 69% of traces are less than 21 instructions long. These traces are likely to schedule quickly without the need of non-optimal heuristics.



(a) (b)

Figure 13: The results for our complete technique on a 3-wide VLIW for (a) code size and (b) run-time, normalized to Bala for a 3-wide VLIW with traces.

9 Conclusions

Code size is an important concern in embedded systems. A method is described for instruction scheduling to reduce code size for a particular subset of VLIWs, without sacrificing run-time. The within-trace scheduler relies on optimal, branch-and-bound methods, as well as heuristics to reduce the compile- time of more-complex traces. Further, we present an across-

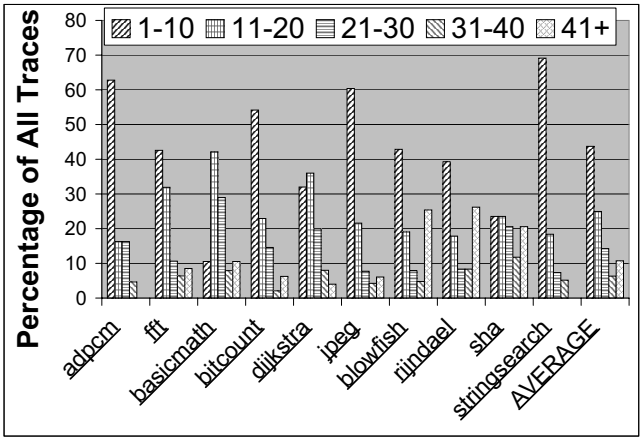


Figure 14: The distribution of trace lengths (not counting NOPS) for each benchmark.

block analyzer that runs before within-trace scheduling; it prevents any instruction from migrating to a different basic block if that move is likely to increase code size. Finally our method is configurable to target either only code size or only run-time, rather than both; this is useful in the least-frequent and most-frequent traces, respectively.

The intellectual novelty of our scheme is seen in the following four contributions. First, we are the first to develop an instruction scheduler that uses back-tracking techniques to minimize code size, and to develop a series of innovative pruning techniques unique to such a search. Second, unlike existing schedulers, our method places constraints on the migration of instructions between blocks – prior to scheduling the trace – whenever this migration is likely to increase code size. Third, in order to approximately identify these code-size-increasing migrations, our method completes a preliminary schedule of basic blocks. Fourth, our method is unique in targeting different objectives for traces of different frequencies – code size only for infrequent traces; run-time only for frequent traces; and both code size and run-time for traces of intermediate frequency.

Our approach improves code size by an average of 16% for a 6-wide and 14% for a 3-wide VLIW, compared to existing trace-scheduling methods, yet retains most of the speedup of trace scheduling.

References

[1] S. Aditya, S. Mahlke, and B. R. Rau. Code Size Minimization and Retargetable Assembly for Custom EPIC and VLIW Instruction Formats. *ACM Trans. on Design Automation of Electronic Systems*, 5(4):752–773, Oct 2000.

[2] D. August, D. Connors, S. Mahlke, J. Sias, K. Crozier, B. Cheng, P. Eaton, Q. Olaniran, and W. Hwu. Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture. In *Proc. of 25th Int'l Symp on Comp Architecture*, July 1998.

[3] V. Bala and N. Rubin. Efficient Instruction Scheduling Using Finite State Automata. In *Proc. of the 28th Annual Int'l Symposium on Microarchitecture (MICRO-28)*, Ann Arbor, Michigan, USA, Nov. 1995. IEEE Computer Society.

[4] S. J. Beaty. Genetic Algorithms and Instruction Scheduling. In *Proc. of the 24th Annual Int'l Symposium on Microarchitecture (MICRO-24)*, pages 206–211, Albuquerque, New Mexico, USA, Nov. 1991. IEEE Computer Society.

[5] J. Bharadwaj, K. Menezes, and C. McKinsey. Wavefront Scheduling: Path Based Data Representation and Scheduling of Subgraphs. In *Proc. of the 32nd Annual ACM/IEEE Int'l Symposium on Microarchitecture*, pages 262–271, Nov 1999.

[6] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing*, 21(2):151–166, June 1999.

[7] H. C. Chou and C. P. Chung. An Optimal Instruction Scheduler for Superscalar Processors. *IEEE Transactions on Parallel and Distributed Systems*, pages 303–313, Mar. 1995.

[8] S. Debray and W. Evans. Profile guided code compression. In *PLDI '02*, June 2002.

[9] E.A.Lee and D. Messerschmitt. Synchronous dataflow. *Proc. of the IEEE*, Sept. 1987.

[10] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: A Technology Platform for Customizable VLIW Embedded Processing. In *Proc. of the 27th Int'l Symp. on Computer Architecture (ISCA)*, Vancouver, Canada, June 2000.

[11] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.*, C-30(7):478–490, July 1981.

[12] S. Freudenberger, T. Gross, and P. Lowney. Avoidance and Suppression of Compensation Code in a Trace Scheduling Compile. *ACM Transactions on Programming Languages and Systems*, 16(4):1156–1214, July 1994.

[13] G. R. Beck and D. W. Yen and T. L. Anderson. The cydra 5 mini-supercomputer:architecture and implementation. *The Journal of Supercomputing*, 7(1):143–180, 1993.

[14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proc. of IEEE 4th Annual Workshop on Workload Characterization*, Dec. 2001.

[15] S. Hanono and S. Devadas. Instruction selection, resource allocation and scheduling in the AVIV retargeting code generator. In *Design Automation Conference*, June 1998.

[16] W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing*, 7(1):229–248, Jan 1993.

[17] S. Jee and K. Palaniappan. Performance Evaluation for a Compressed-VLIW Processor. In *SAC 02*, Madrid, 2002.

[18] D. Kastner and S. Winkel. ILP-based Instruction Scheduling for IA-64. In *Proc. of the ACM SIGPLAN Workshop on Languages*, Snowbird, Utah, USA, June 2001.

[19] S. Y. Larin and T. M. Conte. Compiler-driven cached code compression schemes for embedded ILP processors. In *Proc. of the 32nd Annual ACM/IEEE Int'l Symposium on Microarchitecture on MICRO-32*, Haifa, Israel, Nov. 1999.

[20] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang. Code Optimization Techniques for Embedded DSP Microprocessors. In *32nd Design Automation Conference*. ACM/IEEE, 1995.

[21] J. Liu and F. Chow. A Near-Optimal Instruction Scheduler for A Tightly Constrained, Variable Instruction Set Embedded Processor. In *Proc. of the ACM 3rd Int'l Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES)*, Oct. 2002.

[22] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proc. of the 25th Int'l Symposium on Microarchitecture(MICRO-25)*, 1992.

[23] S. S. Muchnick. *Advanced Compiler Design and Implementation*. M. Kaufmann, San Francisco, CA, 1997.

[24] J. Park and M. Schlansker. On Predicated Execution. Tech. Rep. HPL-91-58, Hewlett Packard Laboratories, Palo Alto, CA, May 1991.

[25] P. Paulin, C. Liem, M. Cornero, F. Nacabal, and G. Goossens. Embedded Software in Real-Time Signal Processing Systems: Application and Architecture Trends. *Invited paper, Proc. of the IEEE*, 85(3), Mar. 1997.

[26] B. Rau, D. Yen, W. Yen, and R. Towle. The Cydra 5 departmental supercomputer. *IEEE Computer*, 22(1):12–35, Jan. 1989.

[27] *The SGI Pro64(TM) compiler suite*. SGI Corporation, March 2000. http://oss.sgi.com/projects/Pro64/.

[28] P. Song. Demystifying EPIC and IA-64. *Microprocessor Report*, 12(1):21, January 26 1998.

[29] A. Suga and K. Matsunami. Introducing the FR500 Embedded Microprocessor. *IEEE Micro*, pages 21–27, Jul 2000.

[30] P. H. Sweany and S. J. Beaty. Dominator-path scheduling: a global scheduling method. In *Proc. of the 25th Int'l Symposium on Microarchitecture*, 1992.

[31] *TMS320C6000 Programmer's Guide*. Texas Instruments, 2003. Also available at www.ti.com.

[32] *Philips Trimedia Processor Home Page*. Philips Corporation, 2002. www.semiconductors.philips.com/trimedia/.

[33] *The TigerSHARC Processor Family*. http://www.analog.com/processors/processors/tigersharc/.

[34] K. Wilken, J. Liu, and M. Heffernan. Optimal Instruction Scheduling Using Integer Programing. In *Programing Language Design and Implementation*, pages 121–133. ACM SIGPLAN, 2000.

[35] T. Zeitlhofer and B. Wess. Code Optimization for the Carmel DSP-CORE. In *Proc. of the Int'l Conference on Signal Processing Applications and Technology (ICSPAT 99)*, Orlando, FL, November 1999.