Hoisting Branch Conditions – Improving Super-Scalar Processor Performance

Bill Appelbe[1], Sri Doddapaneni[1], Reid Harmon[1], Phil May[2], Scott Wills[2], and Maurizio Vitale[1]

[1] College of Computing
[2] School of Electrical and Computer Engineering Georgia Institute of Technology, Atlanta, GA 30332

Abstract. The performance and hardware complexity of super-scalar architectures is hindered by conditional branch instructions. When conditional branches are encountered in a program, the instruction fetch unit must rapidly predict the branch predicate and begin speculatively fetching instructions with no loss of instruction throughput. Speculative execution has a high hardware cost, is limited by dynamic branch prediction accuracies, and does not scale well for increasingly super-scalar architectures.

The conditional branch bottleneck would be solved if we could somehow move branch condition evaluation far forward in the instruction stream and provide a new branch instruction that encoded both the source and target address of a branch. This paper summarizes the hardware extensions to support just such a Future Branch, then gives a compiler algorithm for hoisting branch evaluation across many blocks. The algorithm is applicable to other optimizations for parallelism, such as prefetching data.

1 Introduction

Studies have shown that from 1.6% to 22% of instructions executed are branches or conditional branches [3], with most non-scientific programs much closer to 22%. A branch instruction interrupts the instruction pipeline unless the branch target address is known before the branch instruction is decoded, and the direction of a conditional branch is known. Branch caches enable the branch target address to be determined with fairly high probability at a modest hardware cost [8]. By contrast, determining the direction of a conditional branch is far more difficult. The most common technique is *speculative execution*, in which the direction of the branch is predicted either statically (a bit in the branch instruction) or dynamically (using a cache of previous branch directions). Most modern microprocessors use dynamic prediction. Studies of the prediction accuracy of processors with sophisticated dynamic branch prediction show prediction accuracy of around 90% (e.g., Ultra SPARC [10]: SPECint 88%, SPECfloat 94% [10]; Power 620 [8]: SPEC composite 90% [9]).

Our simulation studies have shown that a fundamental problem with speculative execution is that the overhead of wasted cycles, due to incorrect prediction limits processor throughput, even assuming no data dependencies and unlimited functional units.

Intuitively, conditional branches present a seemingly insuperable bottleneck to increasing superscalarity. Assuming that conditional branches are a fixed proportion of the instructions fetched, the penalty for incorrect branches, and the depth to which branches must be predicted, rises approximately linearly with the degree of superscalarity.

Our insight is that in fact, at the source level, most branches are predictable well before we reach them. In a classic `for loop` a programmer can easily determine whether the next iteration will be taken at the start of the current iteration. If the for loop test is `I <= N`, then if `I <= N-1` at the start of the iteration we know that the next iteration will be taken. The "intelligent programmer" is thus making deductions about what a *future branch* will be. It does not matter how long the loop is, or whether it contains other branches, we can determine at the start of the loop if the next iteration will be taken. To support this at the hardware level, we need a branch instruction that encodes both the source and target address.

To implement future branches work we need:

1. An architectural design of future branch hardware
2. Compiler algorithms that can hoist branch decisions

This paper first summarizes the instruction set design and architecture extensions needed to support future branches (hoisting branch decisions). Then we discuss in detail algorithms for converting regular conditional branches into future branches. Finally, we summarize the implementation status.

It is worth noting that hoisting branch decisions has a rather broader applicability than just compiling for future branches. A common optimization for NUMA parallel processors is prefetching data. We obviously want to avoid prefetching data that will not be needed (to avoid bus contention, cache pollution, and unnecessary instructions). Prefetch instructions are normally issued dozens of instructions before the data will be accessed. To be able to predict whether data will be needed, we need to be able to hoist branch decisions.

2 Future Branch Architecture

Future branch instructions are applicable as extensions to almost any instruction set. We have designed hardware implementations for the Motorola 88010[7], the Power PC 604[8], and the DLX[6]. Future branches bear some superficial similarity to delay slots[6, page 269]: instructions following a branch that are always executed. However, in practice it is hard to fill more than a couple of delay slots, and modern super-scalar microprocessors rely on speculative execution instead of delay slots. From a compiler viewpoint, the algorithm for filing delay slots is simply to find instructions in the basic block contain the branch, or its immediate successor blocks, that have no dependences on the branch. By contrast, the algorithm for hoisting branches tries to move the branch, and all its dependent instructions, forward to another basic block in the flowgraph.

The idea of decoupling the branch decision from the branch itself has been used in many architectures, ranging from those which have separate instructions for computing the branch condition to branch instructions with a variable number of delay slots[4]. However, these all have the limitation that the path and number of instructions executed between the computation of a branch condition and the branch itself is fixed and must be known at compile time. This usually limits hoisting the evaluation of the branch condition to the same basic block as the branch.

With modern superscalar architectures, and non-numerical applications, it is almost invariably necessary to calculate the branch condition in a different basic block than the branch is taken, to avoid the penalty of stalls or speculative execution. Hence future branches, together with hardware support for a queue[3] of pending branches, is needed.

2.1 Future Branch Semantics

Any conditional branch instruction can be represented as:

`S..branch`: **branch** *condition*, `L..target`

where the branch to the target label `L..target` is taken if the *condition* is true. The condition might be a condition code, a register test, a comparison of two registers, and so on. A future branch instruction has the form:

fbranch *condition*, `S..branch`, `L..target`

Its semantics are as follows: upon execution of the instruction, if *condition* is true, then a future branch is *pending*. This means that when the program counter reaches `S..branch` it should change to to `L..target` (i.e., jump).

It turns out to be useful to be able to cancel a pending future branch. For example: a branch may be pending for the next iteration of a loop before we encounter a loop exit. Hence we provide an *unbranch* instruction:

unbranch `S..branch`

We also provide unconditional future branches, and the ability to save/restore pending future branches. Pending future branches are part of the processor state[4], and hence must be saved and restored upon context switches[5]. Finally, we allow more than one future branch to be pending for the same address. Our experimental results show that this is very useful in small loops that have not been unrolled.

---

[3] Actually, a queue cannot be used to hold pending future branches, as they are not necessarily issued in the order that they are taken

[4] There is no indication in the object code, at the branch source, that this is the source of a branch! It is possible to implement future branches that are purely advisory[2], but this has a high space overhead.

[5] By context switches, we mean process context switches rather than procedure calls

3 Future Branch Hardware

An obvious question is whether 16 bits, the usual field size for a branch target, is enough to encode both the source and target address for a branch. Since all branch addresses that we are interested in are PC relative, we first need to determine the distribution of branch distances for a range of benchmarks. Our analysis of a suite of C benchmarks using a modified *gcc* indicates that from .2% (espresso) to 5% (gcc, spice) of branches are for a relative distance of more than 256 instructions. Hence both the relative source and target address can be encoded in the usual 16 bit displacement[6]

The implementation of future branches for the 604 is given in detail in Appelbe et al[2]. Intuitively, the hardware maintains a *future branch cache*, of pending future branches. When the program counter is advanced to fetch instructions, it is compared against branch source addresses in the future branch cache. If any of these is a hit, the contents of the program counter is replaced by the branch target address (i.e., a jump takes place), instructions are fetched from the new address, and the entry is deleted from the future branch cache.

In general, the predicate may not be known until several cycles after the future branch is decoded. This is because the future branch may depend on flags or registers set by an instruction which has not completed. If the predicate is not resolved when the branch source is reached, the branch prediction mechanisms of the PowerPC 604 (branch target address cache, branch history table) are used to best predict the branching path. In an implementation where speculative execution is not provided, instructions fetched using predicted rather than computed predicates would stall in the dispatch stage until the predicate is resolved.

The estimated increase in transistor count for this implementation is 30,000. This does not include hardware already present in the PowerPC 604 (e.g., branch target address cache). For the current PowerPC 604 die, this future branch implementation would occupy just 3.3% of the die area.

4 Compiler Algorithms for Hoisting Branches

Compiling for future branches reduces to the following problem: given a branch instruction `br`, can we *hoist*[1, page 714] this instruction so that at least `N_PIPELINE` instructions lie between the *execution* of the future branch and its target. The value of `N_PIPELINE` is chosen so to "almost guarantee"[7] that the future branch instruction will have completed before the branch target is reached. The parameter `N_PIPELINE` is hardware dependent and branch dependent. For the Power PC 604, a rough upper bound for `N_PIPELINE` is 15. We do not want

[6] Preliminary studies suggest the use of 8 bits signed for both the source and target offsets. The future branch instruction is often far from the branch source address in large loops. Obviously, large scale benchmarking might lead to a different division of the instruction space.

[7] "almost guarantee" implies that we may need to be optimistic about cache misses which cause unusual stalls.

to hoist branches much further than N_PIPELINE, as hoisting too far uses up resources (entries in the branch cache). However, the places where we can hoist branches to depends upon data flow and dominance analysis.

The standard use of code hoisting is to save space. For example: by moving hoisting common subexpressions out of the `then` and `else` parts of a conditional. Our use of code hoisting is quite different, and has more in common with software pipelining. The standard algorithm for code hoisting will not hoist multiple statements at once, or hoist to multiple points of the program. Both of these turn out to be necessary for hoisting branches.

Firstly, we consider the problem of hoisting a single branch. We give a general algorithm and feasibility test for hoisting any branch, then give a global algorithm for choosing branches to hoist and points to hoist them to. Finally, we point out pathological cases where we cannot achieve any significant performance improvement by hoisting branches.

Superficially, the problem of hoisting future branches resembles that of instruction scheduling: reordering machine instructions to maximize thruput. However, there is a fundamental differences

- Branch hoisting is conceptually a source level transformation, whereas instruction scheduling is done at the machine level

4.1 Hoisting a Single Branch

Branch hoisting is a rather curious program transformation that straddles both source and machine level optimization. It is possible to determine the legality of branch hoisting at the source level, but it is difficult to express the transformation in conventional source level representations such as abstract syntax trees. Hence we perform branch hoisting on a low-level form of intermediate code (such as *gcc*'s RTX), after most standard local and loop optimizations, but before physical register allocation or instruction scheduling[8].

The standard algorithm for hoisting evaluation of an expression is to find a point at which an expression is *very busy*[1, page 713] (the operands of the expression are not subsequently defined). In practice, this needs to be modified for hoisting branches, as a branch may need to hoisted to more than one location.

Statement Hoist Criterion: A statement s can be hoisted form a point p to a set of points $\{p_0, p_1, ...p_N\}$ in the program provided:

strong domination on any path from the start of the program to p or from p to itself, that does not including p, passes *exactly once* through one and only one of the points in $\{p_0, p_1, ...p_N\}$.[9]

[8] Instruction scheduling must be aware of future branches, to try to ensure that no stalls occur due to a future branch that is still in the instruction pipeline

[9] Gupta[5], introduced *generalized dominators*: a set of nodes that dominate a point p. However, our *strong dominators* are "stronger", as we require uniqueness, and consider paths from p to itself. We need this as branches must be computed exactly once on each run time path to p.

availability after each point in $\{p_0, p_1, ...p_N\}$, there is no definition of the variables used in s , or use of any variable defined by s .

For branch statements that are based upon condition codes, the condition code is an anonymous variable used by the branch, and set by prior statements.

Algorithms for determining domination and availability are well known[1]. We have had to strengthen the definition of dominance so that the future branch is always at the same loop level as the branch source itself. Also, a future branch can only be issued once (otherwise there will be two future branches pending for the same jump).

Unfortunately, the above algorithm does not work well in practice. Firstly, many branch instructions are immediately preceded by a compare instruction that sets the condition code used by the branch. We could generalize the definition of a conditional branch to include with it any instruction that sets the condition code, but this is ad hoc. Further, in most loops the modification of the loop control variable comes just before the test. For example, consider the following C code fragment that computes the sum of the squares of the positive elements in a list:

```
/* List is a struct consisting of a next pointer and data */
int sumpos(struct List* np) {
    int sum = 0;
    for(; np; np=np->next) {
        if(np->data > 0)
            sum += np->data * np->data;
        }
    return sum;
}
```

Example 1

Like most loops, the increment of the loop control variable comes just before the test. Thus, the above algorithm would be unable to hoist the test far at all. Hence, we need an algorithm which can hoist the calculation condition as well as the branch.

We need to be able to hoist all statements that contribute to the calculation, while preserving all dependences. Consider the following artificial example:

```
    int sum = 0;
    int list[10]
    for(; i<10; ++i) {
        if(list[i]) {
            sum += list[i];
            ++i;
        }
    return sum;
}
```

Example 2

We want to hoist the increment of `i`, test `i<10`, and future branch to the beginning of the loop iteration[10]. The first problem is that `i` is used in the loop: there is an anti-dependency from the use of `i` (in `list[i]`) to its definition (in `++i`). Such anti-dependencies are easily eliminated by renaming: introduce a new variable `i' = i+1`. A far more troublesome problem is the additional, conditional, assignment to `i`. This is a "show stopper"[11]; in general, if there are definitions of a variable *which do not dominate other uses and definitions*, it is impossible in general to hoist the definition beyond the join of those definitions.

Given this criteria, we can define a general recursive algorithm for hoisting any branch and its associated condition.

Statement Hoist Algorithm:

INPUT: a dependence graph, a branch statement br, and a set of points $\{p_0, p_1, ...p_N\}$ which satisfy the strong domination criterion above.

OUTPUT: a transformed program or failure if the branch cannot be hoisted while maintaining dependences.

Let $\{d_0, d_1, ...d_k\}$ be the transitive closure of the set of definitions along paths $Paths_{\{p_0,p_1,...p_N\}} \to p$ with dependences to br (including true, output, and anti-dependences).
All of these definitions will need to be hoisted if the branch is to be hoisted.
These definitions can be hoisted only if:

- For each dependence $\mathtt{s} \to \mathtt{t}$, the source of the dependence dominates the sink

If this condition does not hold, return FALSE.
Otherwise, topologically sort the definitions, and hoist them in that order using **hoist-statement** and finally the branch itself.
Each definition must be hoisted to all points in $\{p_0, p_1, ...p_N\}$ which dominate that definition.
Return TRUE.

[10] Almost all C compilers convert `while` and `for` loops to the `do ... while` form with the increment and test at the end of the loop, as this eliminates a jump

[11] Not quite, we could pipeline the loop to calculate the condition `list[i]` in the previous iteration, but this is the subject of future work.

hoist-statement:
Assume that the statement is of the form: v:= f($uselist$).
If their are any anti-dependences upon the definition d_i,
  in paths $Paths_{\{p_0,p_1,\ldots p_N\}} \rightarrow p$:
    Create a new temporary variable v_{temp}.
    Replace the statement v:= f($uselist$) by v:= v_{temp}, and
    Hoist the statement v_{temp}:= f($uselist$).
otherwise
    Hoist the statement v:= f($uselist$).

One obvious case for specialized optimization is the common case of a `for` loop, such as:

```
for(; i<10; ++i) {
    // uses of i
    }
```

Example 3

Instead of calculating `i' = i+1` and testing `i'<10`, we can simply test `i<9`.

4.2 Global Algorithms

The above algorithm can be used to hoist branches, but it leaves unspecified what hoist points to choose. Also, it says nothing about unbranching in the case of exits from loops.

The following algorithm was based on hand analysis of the source code for several benchmarks. Our compiler analysis of the benchmark *gcc* indicates that it can be improved by using SSA form to determine join points for multiple definitions (page 13).

The global algorithm for hoisting branches for loops and conditional statements in structured programs can be best understood by considering a skeleton program, which is typical of a `while` loop converted to `do` ... `while` form:

```
    // <- p0
    ...
    if (e1) goto EXIT:  // <- b1
    // <- p1
    do {
        // <- p2
        ...
        if (e2) {       // <- b2
            // <- p3
            ...
        } else  {
            if (e3) ... // <- b3
        }
        // <- p4
        ...
    } while (e1);       // <- b4
EXIT:
    // <- p5
```

Example 4

For each of the conditional branches above, we define the hoist points for the branch used by our algorithm.

Conditional Branches Outside Loops (`b1`) This branch must be hoisted to an earlier point in the subprogram (`p0`) which dominates the branch. If there are few instructions in the subprogram before the branch, then inline subprogram calls if possible.

Conditional Branches Within Loops (`b2`) This branch should be hoisted to point (`p2`) at the start of the loop iteration, provided that there are enough statements between `p2` and `e2`.
If the loop is small, or the conditional statement constitutes almost all of the loop, then there may be insufficient statements. In that case, try hoisting the branch to just *after* the end of the conditional statement (`p4`. For the first iteration, a copy of the branch will be hoisted outside the loop (`p1`). If we hoisting `b2` to `p4`, we will need an unbranch at `p5`, as we always predict the branch for the next iteration (which eventually will not be executed).
In some benchmarks, even this does not give us enough instructions between the future branch source and target. In that case, if no definitions from within the loop reach the branch, then we can issue several branches before the loop is entered. We call this technique *branch pipelining*.

Nested Conditional Branches (`b3`) As with non-nested branches, the points which dominate the branch for the next iteration are `p4` and `p2`. We will need an unbranch at `p3` in any case.

Loop Branches (`b4`) Again, the points which dominate the branch are `p2` and `p4`. If the loop body contains sufficient instructions, choose `p2`. Otherwise, hoist the branch test for the first iteration to `p1`, and the branch test for

subsequent iterations to `p4`. This will require an unbranch at `p5`. Short loops are often candidates for branch pipelining.
If there are any exits from the loop, then we will also require an unbranch at `p5`.

In the case of nested loops, first try to convert all branches in the inner loop to future branches. As noted above, procedure inlining can be used for very short procedures whose first instructions are a conditional branch. Loops with very short shortest paths cause problems if we are trying to hoist a branch more instructions ahead than are in the path. Ideally, the compiler will have done standard loop optimizations earlier to unroll such loops. If not, it is possible to pipeline future branches for subsequent iterations.

Procedure calls within loops with future branches present a problem. Architecturally, we can always provide support to save/restore the future branch state across calls, and we are investigating this in our simulator. In practice, it is probably better to just use future branches in leaf procedures, and inline small leaf procedures to create more opportunities for future branches.

A fundamental characteristic of the hardware is that the size of the future branch cache is bounded, and the compiler *must* guarantee not to overflow this buffer.

5 Implementation

The analysis above shows the feasibility of future branches. Nevertheless, validating the usefulness of future branches requires a fairly sustained research effort:

1. Build, or obtain and modify, an instrumented, reconfigurable super-scalar micro-architecture simulator, that incorporates future branch instructions as well as speculative execution for comparison.
2. Modify a production quality C compiler to perform program transformations and code generation for future branches
3. Systematic benchmarking, using standard large-scale benchmark programs.

These efforts are underway, and the results are encouraging.

5.1 Micro-Architecture Simulator

We have acquired a micro-architecture simulator, for the *DLX* instruction-set[6], from McGill university. The simulator is reconfigurable, and supports speculative execution. We have modifying it to support future branch instructions and instrumentation for these. We have extended the *SuperDLX* architecture and implementation in several ways, primarily by including support for future branches. Fortunately, the *DLX* instruction set has many holes in its opcode space, and hence we were able to easily add future branch instructions corresponding to all the conditional and unconditional *DLX* branches. We have also extended the *SuperDLX* simulator to provide better tracing and configuration of branch hardware.

5.2 Compiling for Future Branches

We are basing our implementation of a compiler to generate future branch instructions upon *gcc*. Our reason for doing so is the support that *gcc* provides for targeting almost all commercial microprocessors (including the *DLX*). Using *gcc* gives us portability should we obtain a commercial micro-architecture simulator. Also, *gcc* supports most standard compiler optimizations. Unfortunately, *gcc* does not provide a clean interface to enable manipulation of its low-level intermediate code *RTX*. Further, *gcc* provides only ad-hoc dataflow analysis.

For this and other projects, we have thus undertaken the somewhat radical approach of defining a new environment and interface to *gcc*'s internals using Scheme (a Lisp dialect).

GiL – Gcc In Lisp The fundamental reason for using Scheme for compiler prototyping is that it is a better language than C for prototyping compiler algorithms. Scheme is interpreted, supports a rich heterogeneous data structures library, pattern matching, powerful macros, and so on. GiL is a scheme environment, in which *gcc* is encapsulated as a function call which returns the intermediate code generated at any stage as a Scheme data structure. We have built an extensible set of Scheme libraries for manipulating the Scheme form of the RTX intermediate code, such as general multi-graphs and call-graphs, and are developing Scheme libraries to support scalar dependence analysis and a simplified version of the algorithm for generating future branches.

6 Benchmarks

Initially, we have been hand-coding assembler for standard benchmark kernels. One commonly used set of kernel benchmarks are the Livermore Loop, which consist of 24 kernel loops from scientific applications. Of these 24 loops, 4 contain conditional statements within loops. We have focused on these benchmarks with conditional statements, as long-running loops with large blocks of straight-line code are notoriously easy to get good performance numbers on. By contrast, loops with conditionals are common in non-scientific applications, and difficult for speculative execution to do well at. However, it is important to note that there is no penalty for using future branches in programs with regular loop structures and highly predictable branches.

The shortest of the Livermore Loops benchmarks with conditional statements within loops is benchmark #24, which calculates the minimum in an array. This type of loop is not atypical, and is clearly problematic for speculative execution. There is just no way to correctly guess whether the next element will be less than the minimum! The inner loop is very tight. After optimizing the use of registers, there are just eight or ten instructions in the inner loop, in the version which does not use future branches.

The primary questions that we are interested in answering is: how does the performance of future branches compare with the "best possible" speculative execution, and how do these results vary as the degree of superscalarity changes?

Table 1 summarizes the results:

Superscalarity[12]	**Speculative Branch Depth**				**Future**	**Percentage**
	2	3	4	∞	**Branches**	**Speedup**
3	578	541	541	541	516	4.8%
4	543	526	490	490	428	14.5%
5	528	514	498	490	428	14.5%

Table 1. Cycles Required to Execute Benchmark LL #24 [13]

SuperDLX uses a fairly sophisticated two-bit dynamic branch prediction strategy, and achieved a 83.2% branch prediction accuracy. Further, there were no stores, or stalls on loads, while speculatively executing: the only overhead was lost cycles due to incorrect predictions.

However, as Table 1 shows, the use of Future Branches resulted in significant performance improvement. Table A also illustrates the need for increasingly deep, and expensive, branch prediction at higher superscalarity. If we limit branch prediction to a a maximum depth of 3, then future branches are at least 4.8% faster than speculative execution. As the degree of superscalarity rise, future branches do better, plateauing 14.5%.

The performance gain using future branches is a little surprising because of the four extra instructions in the inner loop of the future branched version. The extra instructions are needed because the inner condition loop condition is a recurrence[14], and the loop is very short [15].

The performance of the future branch version can be improved by loop unrolling and other techniques. However, recurrences within short loops present a fundamental hurdle: most of the work in the loop is devoted to calculating conditions used in subsequent iterations. Our intuition, and the results in the previous section, suggests that such situations are atypical. To more accurately determine the frequency of such situations will require a long term study of large benchmarks, which is underway.

A simple modification of the above benchmark, without recurrences, calculates the sum of the non-negative elements in the array.

Table 2 summarizes the results:

The results for this benchmark are quite dramatic: the future branch version of the benchmark is around 40% faster.

[14] The test in each iteration on is dependent upon the test result in the previous iteration

[15] The test condition is $\mathtt{x[k]} < \mathtt{xm}$, but to predict ahead two iterations we need a test condition of $\mathtt{x}[k+1] < \mathtt{xm}$ and $\mathtt{x}[k+1] < \mathtt{x}[k]$

Superscalarity[16]	Speculative Branch Depth				Future	Percentage
	2	3	4	∞	Branches	Speedup
3	615	570	570	570	396	43.9%
4	576	570	562	550	376	46.2%
5	559	543	515	515	372	38.4%

Table 2. Cycles Required to Execute Benchmark LL #24'

The *DLX* instruction set is comparable to that of most modern RISC processors, hence we expect our results will carry over to any other superscalar architecture.

6.1 Compiler Analysis

Our initial goal was to implement the algorithm above to detect the feasibility of future branches. We analyzed 27 files in the *gcc* distribution looking for branches whose predicate can be trivially hoisted. A predicate is declared trivially hoistable if all the virtual registers needed for its evaluation are defined *exactly* once within the loop it belongs to.

Table 3 shows the distribution of branches in the files we have analyzed. Only about one seventh of the branches are not trivially hoistable. Many of the non hoistable branches could be hoisted by a better algorithm[17]

Total number of conditional branches	4497
Total number of unconditional branches	2270
Not trivially hoistable	764

Table 3. Branch distribution in GCC

Figure 1 shows the distribution of the predicate size (number of instructions necessary to evaluate a predicate). All the instruction needed for a predicate evaluation need to be hoisted as a whole (or at least the last instructions in the chain must be moved far enough from the branch source), thus the shorter these chains are the easier it is to move them.

[17] Hand inspection has revealed that the reason why most branches are not trivially hoistable is that there one definition is inside a conditional. The branch is usually still hoistable, to a point just after the conditional. To recognize this we need ϕ functions and SSA form, which we are now implementing.

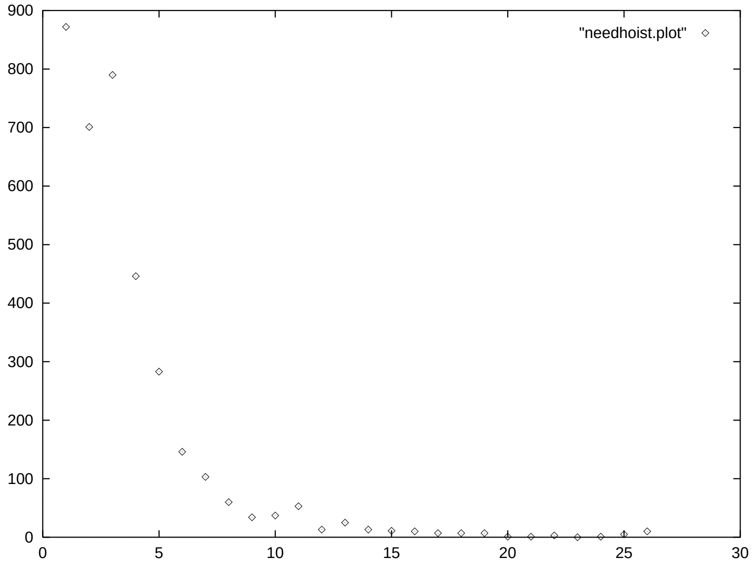

Fig. 1. Distribution of branch predicate sizes

7 Conclusion

Analysis has shown that the traditional approaches to avoiding branch delays, speculative execution, does not scale with increasingly super-scalar architectures. Future branches represent a simple but radical departure from traditional approaches to improving performance of branch instructions. Hardware support for future branches is relatively straightforward to implement. Thus, the feasibility of future branches depends upon largely upon compiler technology to hoist branch decisions sufficiently far. This paper presented an algorithm for doing just that. The algorithm is applicable to another problem in compiling for parallelism: data prefetching. Hoisting branch decisions ahead in the instruction stream uses the same compiler technology that is needed to prefetch data.

Although this paper has demonstrated the feasibility of future branches from architecture, hardware, and compiler viewpoints, there are many open questions and unresolved issues that we are actively addressing at present, including:

- What actual speedups are achieved on standard benchmarks for future branches vs speculative execution?
- What is the interplay between future branches and other compiler optimizations, such as procedure inlining
- Can we use the compiler technology and simulator to investigate the performance improvements that would be available in shared memory multiprocessors that support prefetching?

References

1. Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools.* Addison Wesley Publishing Company, Reading, Massachusetts, 1987.
2. Appelbe, B., Doddapaneni, S., Harmon, R., May, P., Wills, S., and Vitale, M. Future branches – beyond speculative execution. In *Micro-28* ((submitted)).
3. Duvvuru, S., and Arya, S. Evaluation of a Branch Target Address Cache.
4. Goodman, J. R., Hseih, J. T., Liou, K., Pleszkun, A. R., Schechter, P., and Young, H. C. "pipe: A vlsi decoupled architecture". In *Proceedings of The Twelfth Annual Symposium on Computer Architecture* (1989), pp. 20–27.
5. Gupta, R. Generalized dominators and post-dominators. In *Principles of Programming Languages* (San Francisco, California, 1992), p. 246.
6. Hennessy, J. L., and Patterson, D. A. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.
7. Sector, M. S. P. *MC88110 Second Generation RISC Microprocessor Users Manual.* Motorola, Inc., Phoenix, Arizona, 1991.
8. Song, S. P., and Denman, M. The PowerPC 604 Microprocessor. *IEEE Micro* (October 1994), 8–17.
9. Thompson, T., and Ryan, B. PowerPC 620 Soars. *BYTE* (November 1994), 113–120.
10. Wayner, P. SPARC Strikes Back. *BYTE* (November 1994), 105–112.