Speculative Predication

Across Arbitrary Interprocedural Control Flow

H. G. Dietz
Parallel Processing Laboratory
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907-1285
hankd@ecn.purdue.edu

Abstract

The next generation of microprocessors, particularly IA64, will incorporate hardware mechanisms for instruction-level predication in support of speculative parallel execution. However, the compiler technology proposed in support of this speculation is incapable of speculating across loops or procedural boundaries (function call and return). In this paper, we describe compiler technology that can support instruction-level speculation across arbitrary control flow and procedural boundaries.

Our approach is based on the concept of converting a conventional control flow graph into a **meta state** graph in which each meta state represents a set of original states speculatively executed together.

1. Introduction

Speculative execution refers to the concept of executing code before we know if the code will need to be executed. Because this generally has the effect of exposing more instruction-level parallelism, an appropriate architecture might achieve significant speedups — despite the fact that some operations performed speculatively were unnecessary.

However, speculatively executing unnecessary instructions is not always harmless. Speculatively executing some operations, such as stores, could cause the program to produce incorrect results. **Predicates** are simply a way for the compiler to mark speculative instructions so that hardware can directly nullify the operation if the speculated condition proves to be false. In this sense, the word "predicate" is a misnomer; the predicate need not be evaluated before the instruction is issued, but more accurately serves as a **guard** to prevent the instruction from causing potentially harmful effects unless and until the condition is satisfied.

1.1. Then and Now...

Neither the concept of speculative execution nor predication is new. The basic concept of VLIW (Very Long Instruction Word) computers is that compiler technology [Ell85] can be used to move operations from likely future control flow paths up into parallel execution slots within instructions in a dominating block of code; pure hardware speculation is used in most modern microprocessors (e.g., Intel's Pentium III). The concept of explicit predicates or guards appears in the CSP and Occam languages and the SIMD (Single Instruction, Multiple Data) concept of **enable masking**. However, it is

only now that mainstream microprocessors are on the verge of combining both speculation and predication.

For example, Intel's IA64 EPIC (Explicitly Parallel Instruction Computer) instruction set will make extensive use of these features [AuC98]. Speculation is explicitly supported in a variety of ways, with the potential for large instruction-level parallelism widths. There is also a generous set of predicate registers, any of which can be used to guard any instruction, and there are instructions to simplify construction of complex predicates. In summary, compiler technology for predicated speculation is about to become very important.

Unfortunately, compiler techniques for predicated speculation [AuH97] thus far have been somewhat ad hoc. They focus on converting sequences of possibly-nested **`if`** statements into predicates and use of logic minimization techniques to simplify predicate construction.

1.2. Meta State Conversion

In contrast, the approach taken in this paper is to **directly transform a program's conventional state machine description into an equivalent machine description using speculative predicated meta states**. Each meta state consists of a set of guarded original states that can be executed together, and the control flow arcs between meta states behave just like the arcs in the original state machine. We call this process **meta state conversion (MSC)**. Aside from the benefits of having a more formal model for speculative predication, because arbitrary state machines can be given as input, this approach offers the ability to speculate across *arbitrary* control flow and even interprocedural boundaries.

Of course, even this approach is not 100% new. In 1993, we proposed a different type of MSC compilation technique for transforming arbitrary MIMD (Multiple Instruction, Multiple Data) code into pure SIMD (Single Instruction, Multiple Data) code [DiK93]. That technique converted sets of MIMD processing element states that might be executing at the same time on different processors into single, aggregate, meta states representing the complete behavior of the system. Once a program was converted into a single finite automaton based on meta states, that automaton was implemented as pure SIMD code using enable masking to control which processing elements execute which portions of each meta state.

In fact, the way that MSC works for speculative execution is mostly a degenerate case of how it works for MIMD-to-SIMD conversion. In effect, speculation allows a processor simultaneously to be in its current state and one or more potential future states. This is roughly equivalent to generating a SIMD program representing MIMD execution of identical per-processor programs such that some MIMD processing elements logically began execution earlier than others, which also would cause current and future states to coexist.

The good news is that speculative MSC is easier than MIMD-to-SIMD MSC. For example, if there are N processors each of which can be in any of S states, then it is possible that there may be as many as $S!/(S-N)!$ states in the SIMD meta-state automaton. Without some means to ensure that the state space is kept manageable, the technique is not practical. In that MSC is closely related to NFA to DFA conversion, as used in building lexical analyzers, this potential state space explosion is not surprising. However,

for speculative MSC, N=1; the result is that the number of meta states is never more than the number of original states!

This seems strange, since each speculative meta state may contain any of the 2^S possible combinations of original states. However, for speculative MSC, each meta state must begin with precisely **one non-speculative state**, the **core** state, that uniquely identifies it. There are only S potential core states, thus, there cannot be more than S meta states. The speculative MSC algorithm presented in section 3 of the current work confirms this analysis.

1.3. Outline of this Paper

In order to perform a state machine transformation, it is first necessary to express the program as a conventional state machine. The following section describes how this is done for arbitrary programs and gives two examples. Section 3 presents the complete algorithm for speculative MSC, including examples and discussion of how the algorithm can be tuned to be more effective for specific types of target machines. Given a speculative meta state machine, Section 4 briefly overviews the issues involved in generating efficient code for the program. In closing, Section 5 summarizes the contributions of this paper and directions for future work.

2. Construction of the Original State Machine

The conventional state machine version of an original program is created by converting the code into a control flow graph in which each node, or original state, represents a basic block [CoS70]. We assume that these basic blocks are made maximal by a combination of simple local optimizations, removal of empty nodes, and code straightening [CoS70]. However, in order to represent *arbitrary* global and interprocedural control flow, a few tricks are needed.

One might expect that supporting arbitrary global control flow would be a problem, but it is not. In practice, it is most common that each state will have zero, one, or two exit arcs. Zero exit arcs mark termination of the program — for example, a block ending with **`exit(0)`** in a unix C program. Two arcs most often correspond to the **`then`** and **`else`** clauses of an **`if`** statement or to the exit and continuation of a loop. However, constructs like C's **`switch`** or Fortran's computed-**`goto`** can yield more than two exit arcs. For speculative MSC, there is no algorithmic restriction on the number of control flow exit arcs a state may have. Of course, state machines representing loops, even irreducible ones, also are perfectly valid.

```
if (A) {
    do { B } while (C);
} else {
    do { D } while (E);
}
F
```

Listing 1: Simple Example

Listing 1 gives C code for a simple example taken from the MIMD-to-SIMD MSC paper [DiK93]. MSC is primarily a transformation of control flow. Thus, in this example, the specific operations within *A*, *B*, *C*, *D*, *E*, or *F* are not important. It is sufficient to assume that *A*, *C*, and *E* are such that static analysis cannot provide information that would eliminate any of the constructs, e.g., none of these expressions yields a compile-time constant.

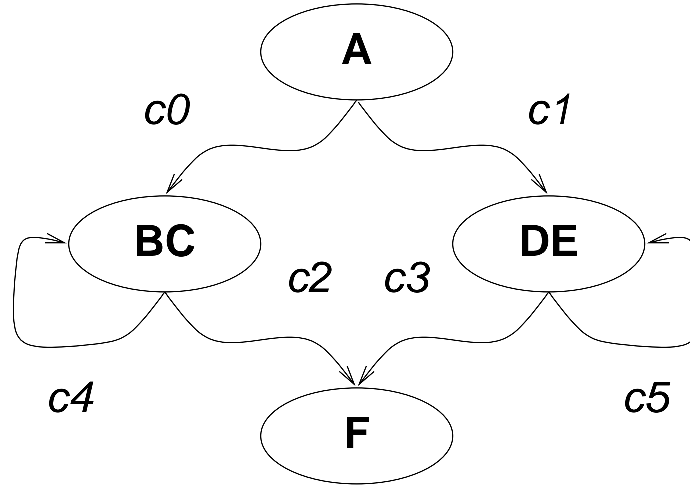

Figure 1: State Graph for Simple Example

The result of mechanically converting this code is Figure 1. Each arc is labeled with the condition under which it is taken. Thus, *c0* corresponds to *A* being true, *c1* corresponds to *A* being false, *c2* corresponds to *C* being false, etc.

Throughout this paper, we will identify predicates by the condition labels from which they are derived. For example, the predicate *5.* would correspond to condition *c5* being used as a guard, and *c5* is really *E* is true, so an instruction guarded by predicate *5.* would have effect only if *E* is true. Since we are tracking blocks of instructions rather than instructions, we would say that, for example, **5.DE** represents all the instructions in node **DE** (i.e., the code from **D** followed by that from **E**) being executed under guard of the predicate *5*. Since speculation can pass through multiple nodes, this would be denoted by using multiple predicates; for example, speculatively executing a second iteration of **DE** at the very start of the program would yield the predicated representation **1.5.DE**.

The tricks are needed to handle interprocedural control flow. In the case of non-recursive function calls, it might suffice to use the traditional solution of in-line expanding the function code (i.e., inserting a copy of the original state graph for the function body). However, to be fully general, our approach should not result in a state graph larger than the sum of the graphs for the component functions, and the approach must be able to handle **recursion**. Our tricks accomplish both goals by simply splicing together separate state graphs — essentially using the fact that `call` and `return` are really just "funny looking `goto`s."

A `call` to a function is, literally, a `goto` accompanied by some stack activity. Because the stack activity is nothing more than manipulation of some data (even if part of that data

looks like a return address), the instructions that implement this activity merely become part of the caller's basic block. Thus, **`call`** becomes an unconditional **`goto`**; literally, a single exit arc from the caller to the first block in the called function. The fact that the **`call`** may be recursive is irrelevant.

Much like **`call`**, a **`return`** also is fundamentally a **`goto`** accompanied by some stack activity that can be placed within a block. However, there is a complication: the target of the **`return`** is not known in the same sense that we know it for a **`goto`**. The trick here is simply to realize that a **`return`** can only **`goto`** one of the sites from which it could have been called. Thus, **`return`** is essentially a somewhat odd-looking **`switch`** statement — a multiway branch — which is perfectly acceptable in a state machine.

```
main(...)
{
 A
  g(...);
 B
  g(...);
 C
}

g(...)
{
 D
  if (E) {
  F
   g(...);
  G
  }
 H
  return;
}
```

```
main:
 A
  goto g;
x:
 B
  goto g;
y:
 C
  exit(...);

g:
 D
  if (E) {
  F
   goto g;
z:
  G
  }
 H
  switch (...) {
  Case x: goto x;
  case y: goto y;
  case z: goto z;
  }
```

Listings 2 and 3: Recursive Function Example and **`goto`**-Conversion

A simple example of this **`call`**/**`return`** processing is given in Listings 2 and 3 and Figure 2. Listing 2 shows two C functions, **`main()`** and **`g()`**. In Listing 3, the **`goto`**-converted version is given; notice that recursion essentially looks like a loop. The resulting original state graph is given in Figure 2.

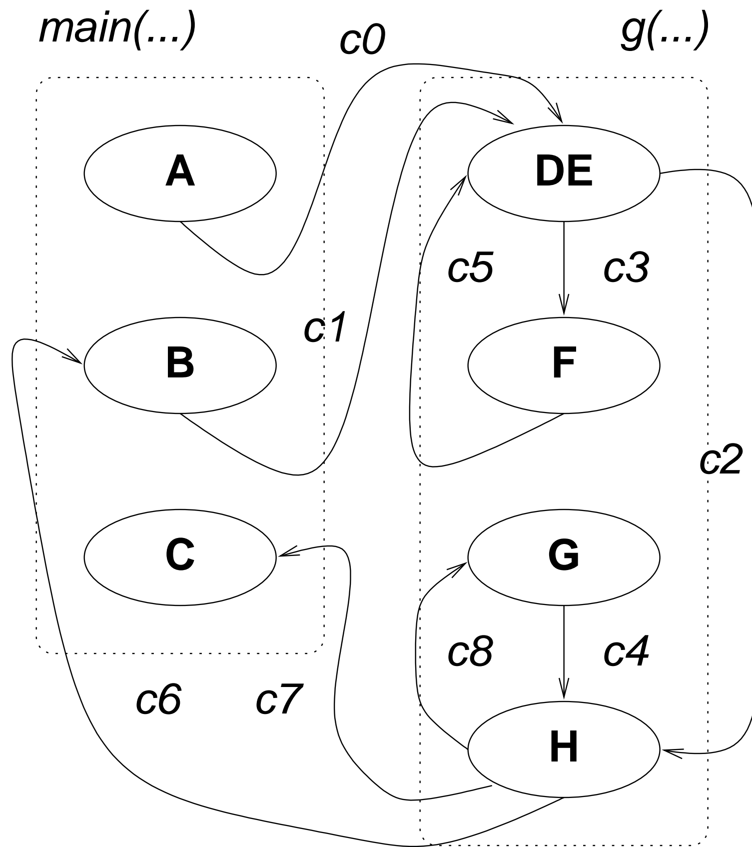

Figure 2: State Graph for Recursive Function Example

For completeness, notice that Figure 2 has every arc labeled with a condition despite the fact that many of these conditions are actually the constant *true*. In particular, conditions *c0*, *c1*, *c4*, and *c5* are all the constant *true*.

There is one additional complication that can arise in constructing the original graph: some states may need to be given the attribute that they are inherently non-speculative. For example, many operating system calls, especially those involving I/O, have this property. These operations are simply encoded as separate states that are marked as non-speculative, with the result that these states will be represented verbatim as meta states in the converted graph. Note that marking operations with the non-speculative attribute requires that they be isolated from any other operations that were in the same basic block, which may require cracking basic blocks into multiple states.

3. Speculative Meta State Conversion

The speculative MSC algorithm is a surprisingly straightforward transformation of the state graph. In large part, this is because the potential state space explosion seen in both MIMD-to-SIMD MSC [DiK93] and NFA-to-DFA conversion simply cannot occur when performing speculative MSC. Still, there are a few new problems that the speculative

MSC algorithm must address and a multitude of new performance-tuning issues.

Like most state-space conversions, our algorithm begins with the start state. Clearly, the start state of the original program is also the non-speculative **core** of the start meta state for the transformed program. In speculative execution, potential future states from the original program graph can be executed simultaneously with the start state, so our next step is a "closure," which we will view as the recursive application of "reaching," to add predicated original states into the new start meta state. As this meta state is constructed, if we choose not to merge a particular reachable state into this meta state, that state is instead added to a work list of meta states remaining to be processed, and an appropriate arc is added to the meta state graph. This same procedure is then applied to process each meta state until the work list has been emptied. This process is as straightforward as it sounds.

The following C-based pseudo code gives the base algorithm for speculative predicated meta-state conversion:

```
meta_state_convert(state start)
{
 state s;
 metastate m;

 /* Make a node for the meta state
    that will be the start state...
    it has the original start as core
 */
 make_core(start);

 /* While there are unmarked cores */
 while ((s = get_umarked_core()) != ∅) {
  /* We are taking care of this core */
  mark_core(s);

  /* Create a metastate name */
  make_state_name(s);

  /* Compute speculative closure */
  m = {true.s};
  m = reach(s, m, true, s, maxdepth);

  /* Optimize predicates in m using
     logic minimization, etc.
  */
  m = optimize_predicates(m);

  /* Insert meta state contents */
  make_state_contents(s, m);
 }
}
```

Of course, the real work is done in the **`reach()`** algorithm, which is applied recursively to speculatively merge states. Wisely choosing which states to merge into each meta state is a very difficult problem. In the general case, each original state may have any number of exit arcs. If there are k successor states for the start state, there are 2^k different ways in which the successors could be merged for speculative execution. Of course, each of the successor states may have one or more successors as well, and these states also may be merged for speculative execution. Thus, we see an explosion not in the number of meta states constructed, but in the potential complexity within each meta state.

How extensive should speculative merging be? Although it might seem that the entire program could be merged into the speculative start state, such merging generally is not possible because cycles in the graph would yield infinitely large meta states. The reason is that program loops require a (speculative) *copy of each instruction for each iteration*, so **`while`** loops that have unbounded iteration effectively would be unwound infinitely. In any case, only a relatively small, machine-dependent, degree of speculation is profitable, so the merging of successor states should be kept appropriate for the target architecture.

For our initial **`reach()`** algorithm, we suggest three very simple rules for limiting speculation:

1. Set a maximum depth, called **`maxdepth`** in **`meta_state_convert()`**, beyond which speculation is not considered. If **`maxdepth`**=0, no speculation is done. If **`maxdepth`**=1, only speculation across single original state transitions will be allowed. Setting **`maxdepth`**=∞ essentially disables this constraint.
2. Because potentially infinite loop expansion would otherwise be difficult to avoid, having multiple copies of an original state within a meta state is disallowed. This has the side benefit of significantly simplifying bookkeeping for code generation.
3. A state that was created with the non-speculative attribute is always a meta state unto itself, never speculatively executed.

Using these three rules, the **`reach()`** algorithm is:

```
metastate
reach(state core, /* core original state */
metastate m,      /* meta state */
pred p,           /* predicate to s */
state s,          /* reach is from s */
int depth)        /* depth from core */
{
 /* Next level of successors */
 foreach (s→x on condition c) {
  /* Is this end of speculation? */
  if (nonspeculative(s) ||
      nonspeculative(x) ||
      (depth < 1) ||
      ({q.x} ∈ m)) {
   /* At an edge of the meta state
      because depth reached or
      state is already included;
      add another meta state core,
      doing nothing if it exists
   */
   make_core(x);

   /* Add meta state exit arcs */
   make_exit_arc(core, p & c, x);
  } else {
   /* Keep speculating on successors */
   m = m ∪ {(p & c).x};
   m = reach(core, m, p & c, x, depth - 1);
  }
 }
 return(m);
}
```

To better understand how the algorithm works, it is useful to return to our examples. Using **`maxdepth`**=∞, the simple example given in Figure 1 becomes the meta state graph shown in Figure 3.

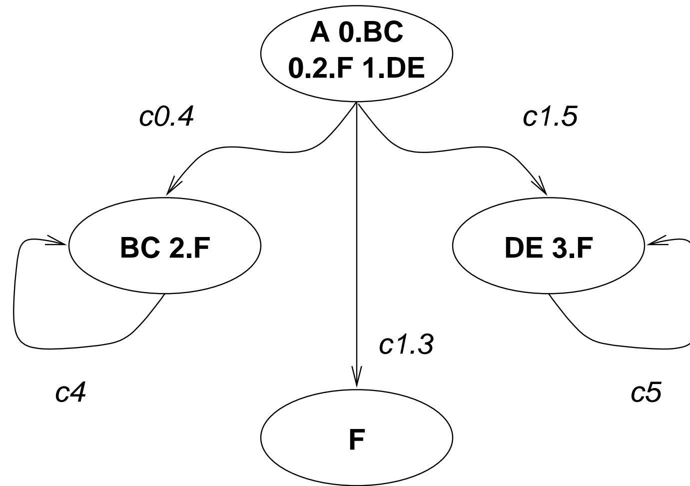

Figure 3: Speculative Meta-State Graph for Simple Example

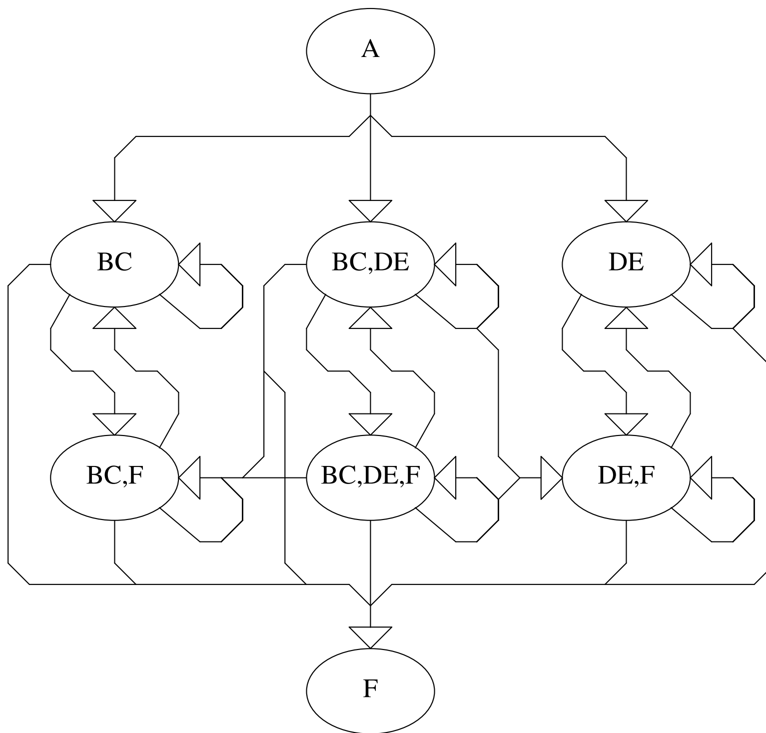

Figure 4: SIMD Meta State Graph for Simple Example

At first glance, it might be somewhat disturbing that **1.3.F** is not also absorbed into the start meta state of Figure 3, instead yielding a separate meta state for **F**. This occurs because **0.2.F** was merged into that state first, thus, the existence of another copy of **F** in the meta state blocked the speculation of **1.3.F** by rule 2. The **1.3.F** speculation does not need to be blocked in the sense that **0.2.F** and **1.3.F** are not involved in detection of a loop, but the ability to optimize **F**'s code based on knowing it came through the **0.2.** path might yield optimizations that would not have been possible if both the **0.2.** and **1.3.** paths were merged to describe a single copy of **F**'s code. In summary, the result may look strange, but we will have to wait for speculative predicated hardware before we can know if performance is damaged.

It is interesting to note that this graph is quite different from that generated using MIMD-to-SIMD MSC [DiK93] on the same input. That graph is given in Figure 4. Notice the increase in the number of meta states: the original graph had 4 states and so does the speculative meta state graph, but the SIMD meta state graph has 8. The concepts may be similar, but the behavior of the two MSC algorithms is clearly very different.

The recursive function example given in Figure 2, when processed using **`maxdepth`**=∞, yields the speculative meta state graph shown in Figure 5.

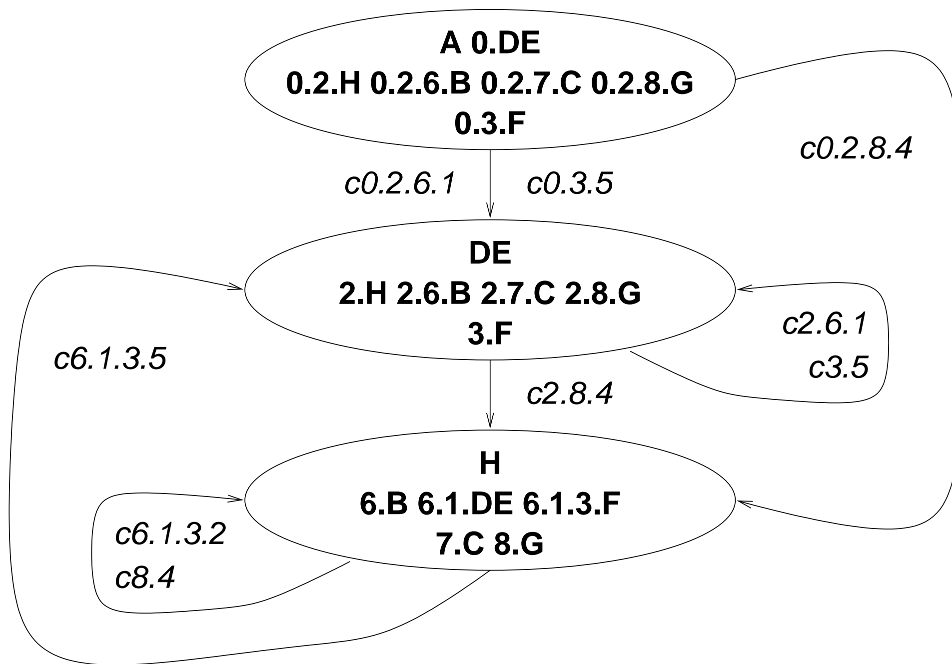

Figure 5: Speculative Meta-State Graph for Recursive Function Example

The original graph containing 7 states (Figure 2) is reduced into a graph with only 3 speculative meta states (Figure 5). However, all but the code for **A** is *replicated in every meta state*. Of course, just because the code started out identical does not imply that it remains so after various local optimizations have been performed; further, if the code is truly identical, it may be possible to improve cache performance and reduce program size by creating a subroutine wrapper for the common code and sharing access to that.

Still, if we ignore optimizations, the basic result is roughly a three-fold increase in program size. Of course, we also have produced speculative "basic blocks" that are on average 6 times larger than the non-speculative basic blocks, so we have dramatically increased the opportunities for instruction level parallelism (ILP).

Depending on a variety of architectural and compiler back-end characteristics, huge blocks might not be better than medium size blocks. For this reason, it is useful to also consider what happens if the recursive function example is processed using the minimum speculation, **`maxdepth`**=1. The resulting graph is given in Figure 6.

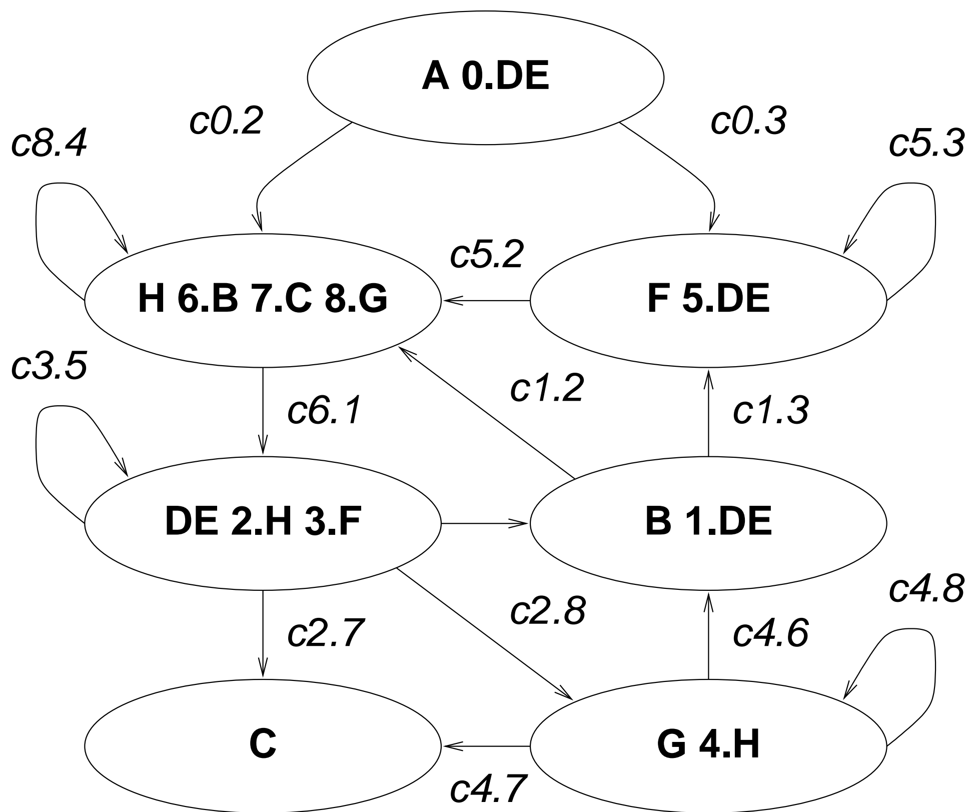

Figure 6: Minimally Speculative Graph for Recursive Function Example

This example is really too small to fully demonstrate the difference, but reducing the value of **`maxdepth`** will generally reduce both the size of typical speculative blocks and the total program size. Interestingly, there are more meta states in the **`maxdepth`**=1 version than in the **`maxdepth`**=∞ version; we do not know for certain if this is a quirk of the example, but suspect that it is not. In effect, using deeper speculation allowed some cores to be omitted because they were always speculated correctly before that core state would have been reached, no matter which paths were taken. We expect this general property to hold for most program state graphs.

In summary, the algorithm as we have presented it offers just one tunable parameter, **`maxdepth`**, and even use of this is somewhat unclear. However, many variations are possible:

- Make individual speculation decisions based on the estimated improvement (cost in terms of time or space) in that particular speculation. E.g., do not speculatively execute block **A** with block **B** unless parallel execution of these two blocks is possible and would be expected to yield speedup over not speculating.
- While performing speculative MSC, if a state is encountered that can be divided into a portion worth speculating and a portion not worth speculating, that state can be split into two states and speculation applied only to the profitable portion. The MIMD-to-

SIMD MSC algorithm used a similar technique to split states where execution time imbalances existed [DiK93].

- Rule 2 for speculative state merging can be augmented by a technique that requires more bookkeeping, but better handles speculation involving loops. For example, by unrolling a loop by a factor of two, we would obtain different states for the loop body such that two iterations could be speculated without violating rule 2. Another possibility would be to replace rule 2 with a technique that prohibits speculative duplication of a state within a meta state *only* when the duplicates could not share the same speculative instructions; i.e., Figure 3 would be changed only in that the **F** meta state would be absorbed into the start meta state, which would then contain **A 0.BC 1.DF (0.2** *or* **1.3).F**.

Experimentation with these variations awaits availability of appropriate target hardware and detailed timing models.

4. Coding of the Meta-State Automaton

Given a program that has been converted into a speculative meta state graph, most aspects of coding the graph are fairly straightforward. The two most interesting issues are the coding of multiway branches and methods for increasing the probability that each speculatively-executed instruction will be useful.

4.1. Multiway Branch Encoding

Although multiway branches are relatively rare in high-level language source code, the state machine conversion process often converts **`return`** instructions into multiway branches. Additional multiway branches can be the direct result of the speculative meta state conversion process itself; for example, state **G 4.H** in Figure 6 essentially inherited one exit arc from original state **G** and two more from **H**.

For conventional machines, the most effective way to code a multiway branch is generally to create a customized hash function that can be used to index a jump table [Die92]. Oddly, predication offers a simpler way that essentially **converts `switch` statements into `return` statements**. A **`return`** is really an indirect `goto` in which the designated storage cell (e.g., a particular register) holds the destination address. Thus, we can implement arbitrary multiway branches by simply applying the appropriate predicates to instructions that place the target value in that storage cell.

For example, consider meta state **G 4.H** in Figure 6. To implement the 3-way branch, code like that shown in Listing 4 should suffice.

```
(4.6).load addr_register, location_of(B 1.DE)
(4.7).load addr_register, location_of(C)
(4.8).load addr_register, location_of(G 4.H)
      goto *(addr_register)
```

Listing 4: Predicated Multiway Branching

4.2. Common Subexpression Induction

The true limit on performance improvement by speculative predication is not how much can be grouped together for parallel execution, but how much *useful* work can be executed in parallel. Speculating with greater depth may increase the number of instructions available for parallel execution, but it also tends to lower the probability that those instructions will be useful, i.e., that they are on the control flow path that the code actually takes.

Because the examples in this paper all describe basic blocks as the fundamental unit of code, it is easy to forget that instructions, not blocks, are the predicated units of execution. The fact that two blocks differ does not imply that they contain wholly different instructions.

Often approximated by SIMD hand coders [NiT90][WiH91], Common Subexpression Induction (CSI) [Die92a] is a compilation technique that, given a set of guarded blocks of code, attempts to rewrite the code to maximize the set of common subexpressions (instruction sequences) that can be shared by multiple, originally disjoint, guards. Although CSI was originally developed as a SIMD optimization designed to allow a larger fraction of the SIMD processing elements to be enabled for each instruction, the exact same technique applied to predicated speculative execution yields an increase in the probability that the the operations being performed are useful.

The SIMD CSI algorithm can be summarized as follows. First, a guarded DAG is constructed, then this DAG is improved using inter-thread CSE. The improved DAG is then used to compute information for pruning the search: earliest and latest, operation classes, and theoretical lower bound on execution time. Next, this information is used to create a linear schedule (SIMD execution sequence), which is improved using a cheap approximate search and then used as the initial schedule for the permutation-in-range search that is the core of the CSI optimization.

Although the SIMD CSI algorithm is usable for speculative predication, we can significantly simplify the algorithm. For SIMD machines, masking (predication) is a “modal” operation; thus, the cost of an instruction sequence is a function of the number of times that masks are changed, which depends on the precise (sequential) instruction order. For speculative predication, only the creation of the guards has cost, so the CSI algorithm need not search all linear instruction orders to determine the best coding. We are currently developing this new predicated variant of CSI.

5. Conclusions

The evolution of a new breed of microprocessors, predicated speculative execution engines, appears to demand a whole new breed of compiler transformations. However, just as neither hardware for predication nor speculation is truly new, the required compiler technology also can be most effectively fashioned from the techniques developed before. The approach detailed in this paper is quite new, and embodies several new algorithms, but is clearly derived from compiler technology developed for MIMD-to-SIMD conversion in 1992-1993.

In this paper, we have demonstrated a simple, fully general, way for a compiler to manage predicated speculative execution spanning arbitrary control flow and procedural boundaries. We anticipate further adaptation and improvement of compiler techniques which were originally developed for SIMD architectures.

References

[AuC98] David I. August, Daniel A. Connors, Scott A. Mahlke, John W. Sias, Kevin M. Crozier, Ben-Chung Cheng, Patrick R. Eaton, Qudus B. Olaniran, and Wen-mei W. Hwu, "Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture," *Proceedings of the 25th International Symposium on Computer Architecture*, July 1998.

[AuH97] David I. August, Wen-mei W. Hwu, and Scott A. Mahlke, "A Framework for Balancing Control Flow and Predication," *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[CoS70] J. Cocke and J.T. Schwartz, *Programming Languages and Their Compilers,* Courant Institute of Mathematical Sciences, New York University, April 1970.

[Die92] H.G. Dietz, "Coding Multiway Branches Using Customized Hash Functions," Technical Report TR-EE 92-31, School of Electrical Engineering, Purdue University, July 1992.

[Die92a] H.G. Dietz, "Common Subexpression Induction," Proceedings of the *1992 International Conference on Parallel Processing,* Saint Charles, Illinois, August 1992, vol. II, pp. 174-182.

[DiK93] H. G. Dietz and G. Krishnamurthy, "Meta-State Conversion," *Proceedings of the 1993 International Conference on Parallel Processing*, vol. II, pp. 47-56, Saint Charles, Illinois, August 1993.

[Ell85] Ellis, J.R., *Bulldog: A Compiler for VLIW Architectures.* MIT Press, Cambridge, MA, 1985.

[NiT90] M. Nilsson and H. Tanaka, "MIMD Execution by SIMD Computers," Journal of Information Processing, Information Processing Society of Japan, vol. 13, no. 1, 1990, pp. 58-61.

[WiH91] P.A. Wilsey, D.A. Hensgen, C.E. Slusher, N.B. Abu-Ghazaleh, and D.Y. Hollinden, "Exploiting SIMD Computers for Mutant Program Execution," Technical Report No. TR 133-11-91, Department of Electrical and Computer Engineering, University of Cincinnati, Cincinnati, Ohio, November 1991.