A Compiler Cost Model for Speculative Parallelization

JIALIN DOU
ARM Ltd.
and
MARCELO CINTRA
University of Edinburgh

Speculative parallelization is a technique that allows code sections that cannot be fully analyzed by the compiler to be aggressively executed in parallel. However, while speculative parallelization can potentially deliver significant speedups, several overheads associated with this technique can limit these speedups in practice. This paper proposes a novel compiler static cost model of speculative multithreaded execution that can be used to predict the resulting performance. This model attempts to predict the expected speedups, or slowdowns, of the candidate speculative sections based on the estimation of the combined runtime effects of various overheads, and taking into account the scheduling restrictions of most speculative execution environments. The model is based on estimating the likely execution duration of threads and considers all the possible permutations of these threads. This model also produces a quantitative estimate of the speedup, which is different from prior heuristics that only qualitatively estimate the benefits of speculative multithreaded execution. In previous work, a limited version of the framework was evaluated on a number of loops from a collection of SPEC benchmarks that suffer mainly from load imbalance and thread dispatch and commit overheads. In this work, an extended framework is also evaluated on loops that may suffer from data-dependence violations. Experimental results show that prediction accuracy is lower when loops with violations are included. Nevertheless, accuracy is still very high for a static model: the framework can identify, on average, 45% of the loops that cause slowdowns and, on average, 96% of the loops that lead to speedups; it predicts the speedups or slowdowns with an error of less than 20% for an average of 28% of the loops across the benchmarks and with an error of less than 50% for an average of 80% of the loops. Overall, the framework often outperforms, by as much as 25%, a naive approach that attempts to speculatively parallelize all the loops considered, and is able to curb the large slowdowns caused in many cases by this naive approach.

A preliminary version of this research was described in the *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT 2004)*.
Work conducted in part while the first author was with the University of Edinburgh. This work was supported in part by EPSRC under grant GR/R65169/01.
Author's address: Jialin Dou, ARM Ltd., 110 Fulbourn Road, Cambridge, CB1 9NJ, UK; Marcelo Cintra, University of Edinburgh, Edinburgh, United Kingdom.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors

General Terms: Measurement, Performance

Additional Key Words and Phrases: Speculative parallelization, speculative multithreading, thread-level speculation

ACM Reference Format:

Dou, J. and Cintra, M. 2007. A compiler cost model for speculative parallelization. Architec. ACM Trans. Code Optim. 4, 2, Article 12 (June 2007), 36 pages. DOI = 10.1145/1250727.1250732 http://doi. acm.org/10.1145/1250727.1250732

1. INTRODUCTION

As exploitation of greater degrees of instruction level parallelism seems to provide diminishing gains, thread-level parallelism becomes a more attractive choice, especially considering the current trend toward chip multiprocessor (CMP) systems. Unfortunately, explicit parallel programming is not yet commonplace and parallelizing compilers still fail to parallelize a significant set of codes when data-dependence information at compile time is incomplete. To aid in the parallelization process, hardware support for *speculative parallelization* (also known as *thread-level speculation* or *speculative multithreading*) has been proposed [Akkary and Driscoll 1998; Dubey et al. 1995; Hammond et al. 1998; Krishnan and Torrellas 1999; Marcuello et al. 1998; Ooi et al. 2001; Sohi et al. 1995; Steffan and Mowry 1998; Tsai et al. 1999]. In this approach, potentially dependent threads are speculatively executed in parallel and hardware mechanisms monitor the memory reference stream to detect and correct any data-dependence violation.

While speculative parallelization can potentially deliver significant speedups for code sections that would otherwise be executed sequentially, several overheads associated with the technique limit these speedups in practice. In fact, in many cases, these overheads can lead to slowdowns with respect to a sequential execution. Thus, accurately identifying, and quantifying, these overheads is critical for good overall performance. Five major sources of overheads in speculative parallelization have been identified [Ooi et al. 2001; Oplinger et al. 1999; Vijaykumar 1998; Vijaykumar and Sohi 1998]: thread *squash and restart* resulting from data-dependence violations, speculative *buffer overflow*, *load imbalance*, thread *dispatch and commit*, and interthread *communication*.

Current compiler technology for speculative parallelization is still maturing. Static compiler analyses to select speculative threads are still based on simple heuristics and only indirectly estimate the speculative multithreaded execution overheads [Bhowmik and Franklin 2002; Chen et al. 2003; Kim and Eigenmann 2001; Li et al. 1996; Vijaykumar and Sohi 1998; Zhai et al. 2004]. These heuristics also tackle individual overheads and there is no integrated approach to jointly consider all overheads. Finally, these heuristics only provide a qualitative prediction of the suitability of code sections for speculative multithreaded execution. Most recent compiler efforts have relied on profiling for thread selection [Liu et al. 2005; Quinones et al. 2005]. Profiling complements static analyses, but is not always applicable.

The main contribution of this paper is to present a model of speculative multithreaded execution that can be used by the compiler to reason about the overheads and expected resulting performance gains, or losses, from speculative parallelization. This model attempts to predict the expected speedups, or slowdowns, of the candidate speculative sections based on the estimation of the combined runtime effects of various speculation overheads, and taking into account the scheduling restrictions of most speculative execution environments. The model is based on estimating the likely execution duration of threads and considers all the possible permutations of these threads. Where compile-time information is incomplete, the model can be easily parameterized to use runtime or profile information. The output of the model is not a simple qualitative prediction of whether a given section of code is "good" for speculative multithreaded execution, but rather a quantitative prediction of the actual speedup or slowdown. This knowledge can assist the compiler or runtime system to make more complex and educated tradeoff decisions. For instance, in a highly loaded multiprogrammed environment the compiler or runtime system may decide to switch off speculative parallelization, even when a speedup is expected, if this speedup is too small and does not justify the use of the extra resources.

The proposed cost model was implemented in the SUIF research compiler development framework. The resulting framework was used on a collection of SPEC benchmarks to estimate the impact of load imbalance, thread dispatch and commit, and squash and restart overheads on the resulting speedups of speculative parallelization. The framework was also found to be very stable and efficient with moderate compilation times. Experimental results show that the framework can identify, on average, 45% of the loops that cause slowdowns and, on average, 96% of the loops that lead to speedups. In fact, the framework predicts the speedups or slowdowns with an error of less than 20% for an average of 28% of the loops across the benchmarks and with an error of less than 50% for an average of 80% of the loops. Compared to a limited version of the framework that did not consider squash and restart overheads [Dou and Cintra 2004], the accuracy when loops with violations are included is lower, but still very high for a static model. Overall, the framework often outperforms, by as much as 25%, a naive approach that attempts to speculatively parallelize all the loops considered, and is able to curb the large slowdowns caused, in many cases, by this naive approach.

The rest of the paper is organized as follows: Section 2 describes speculative parallelization and its sources of overheads. Section 3 presents our proposed compiler framework for estimating the impact of various overheads on the performance of speculative parallelization. Section 4 describes our compilation infrastructure and our evaluation methodology. Section 5 presents the experimental results. Section 6 discusses related work; and Section 7 concludes the paper.

2. SPECULATIVE PARALLELIZATION

2.1 Execution Model

Under the *speculative parallelization* (also called *thread-level speculation* or *speculative multithreading*) approach, sequential sections of code are

speculatively executed in parallel, hoping not to violate any sequential semantics. The control flow of the sequential code imposes a total order on the threads. At any time during execution, the earliest thread in program order is *nonspeculative*, while the others are *speculative*. The terms *predecessor* and *successor* are used to relate threads in this total order. Stores from speculative threads generate unsafe *versions* of variables that are stored in some sort of *speculative buffer*, which can be simply the private caches or some additional storage dedicated to speculative parallelization. If a speculative thread overflows its speculative buffer it must stall and wait to become nonspeculative (or be squashed). Loads from speculative threads are provided with potentially incorrect versions. As execution proceeds, the system tracks memory references to identify any cross-thread data-dependence violation. If a dependence violation is found, the offending thread must be *squashed*, along with its successors, thus reverting the state back to a safe position from which threads can be reexecuted. In addition to this implicit memory communication mechanism, some hardware environments for speculative parallelization allow synchronized *memory and register communication* between neighbor threads. When the execution of a nonspeculative thread completes, it *commits* and the values it generated can be moved to safe storage (usually main memory or some shared higher-level cache). At this point, its immediate successor acquires nonspeculative status and is allowed to commit. When a speculative thread completes it must wait for all predecessors to commit before it can commit. After committing, the processor is free to start executing a new speculative thread. Usually a processor that completes the execution of a speculative thread before the predecessor threads have committed is not allowed to start execution of a new speculative thread.

2.2 Speculative Parallelization Overheads

The execution model of speculative parallelization (Section 2.1) leads to five major overheads: thread *squash and restart* as a result of data-dependence violations, speculative *buffer overflow*, *load imbalance*, thread *dispatch and commit*, and interthread *communication*. The thread *squash and restart* overhead is mainly composed in time to flush the speculative buffers and the redundant reexecution of part of the thread prior to the violation (Figure 1a). This overhead is dictated by the actual frequency of data-dependence violations and by the location of the dependences within the threads. The speculative *buffer overflow* overhead relates to the amount of time the processor remains idle after a speculative thread overflows its speculative buffer and until it is allowed to proceed (Figure 1b). This overhead is dependent on the physical size and organization of the speculative buffer, the amount of data written by the thread, and the size of threads. An associated overhead appears when there is no dedicated speculative buffer and the speculative dirty data is kept, and pinned, in the cache, in which case there may be some degradation in cache performance. The thread *dispatch and commit* overhead is mainly composed in time to move the speculatively modified data from the speculative buffer to safe storage and the time to update the system state to reflect the new status of the threads. This overhead depends mainly on the amount of data written by the thread. The interthread *communication* overhead relates to the time the processor remains

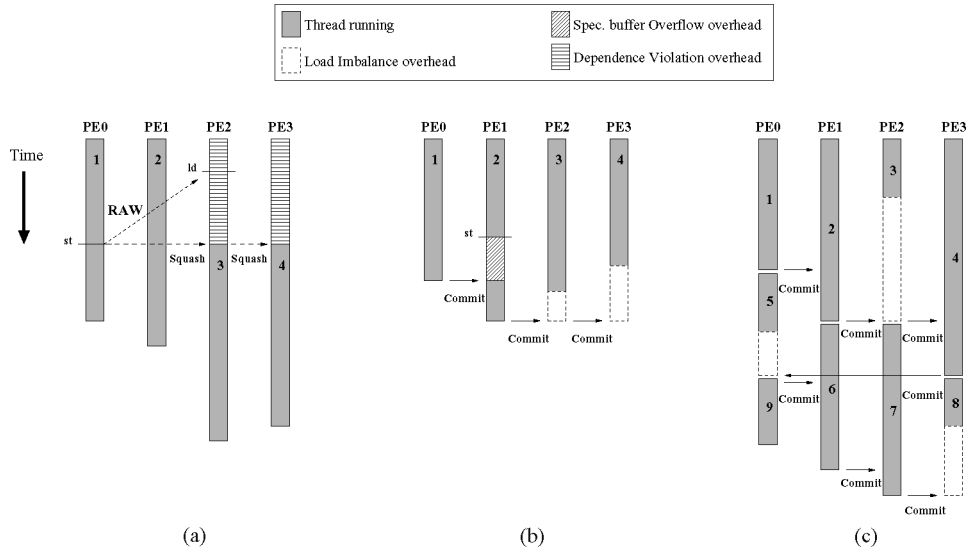

Fig. 1. Some speculative parallelization overheads: squash and restart (a); speculative buffer overflow, also leading to further load imbalance overhead (b); and load imbalance (c). The numbers inside the bars correspond to the original sequential order of the threads.

idle while a speculative thread waits for a memory or register value produced and communicated by a predecessor thread. This overhead is only relevant in architectures that support synchronized register or memory communication and is dependent on the frequency of such communication points and the location of these within the threads. Finally, *load imbalance* overhead relates to the time the processor remains idle after completing the execution of a speculative thread and until this thread becomes nonspeculative (Figure 1b and 1c). This overhead depends mainly on the differences in execution time among threads, which is, in turn, heavily influenced by the other overheads above. We note that the impact of load imbalance in speculative parallelization is far greater than in traditional, nonspeculative, parallelization, because, in the latter, a new thread can be assigned to a processor as soon as the current thread finishes, while with speculative parallelization a processor cannot start work on a new thread until its current thread becomes nonspeculative and commits. Finally, another side-effect overhead of speculative parallelization is a degradation in bandwidth performance on the coherence bus because of the significantly higher traffic generated by the speculation protocol. Of the overheads discussed above, thread squash and restart, speculative buffer overflow, and load imbalance have been identified as the most significant overheads [Ooi et al. 2001; Vijaykumar 1998; Vijaykumar and Sohi 1998].

3. FRAMEWORK FOR MODELING OVERHEADS

In Dou and Cintra [2004], we presented the base thread tuple model and discussed how it could be extended to accommodate all speculative parallelization overheads. Here, we flesh out the extended and complete model.

3.1 Basic Idea

Instead of using a collection of heuristics to identify "good" code sections for speculative parallelization, we propose to compute a quantitative estimate of the actual speedup. With this result, the compiler or runtime system can make an informed decision on whether to try speculative parallelization or to run those code sections sequentially.

To compute the estimated speedup (S_{est}) we use a compiler model of the speculative multithreaded execution that is based on estimated thread sizes and probabilities of occurrence of these sizes on the individual processors. The inputs to the model are the number of processors in the system (P), the possible sizes of threads, including overheads, and their probabilities. The model then considers all possible groupings of thread sizes across the P processors. Each grouping, which we call a *thread tuple*, has a probability of occurrence, which is the joint probability of occurrence of each thread size in the tuple, additionally taking into account the mapping of threads in the tuple to physical processors. Each thread tuple also has a sequential and a parallel execution time, which, together with the probability of occurrence of the tuple, is used to compute the overall estimated sequential ($Tseq_{est}$) and parallel ($Tpar_{est}$) execution times of the average tuple, and, thus, the estimated speedup (S_{est}).

The computation of the possible thread tuples is divided in two parts. First, we generate only thread sizes that appear through possible execution of different control paths. These thread sizes are then intrinsic to the code structure and do not include speculative execution overheads. We call these the *base thread sizes* and, with them, we generate the *base thread table*. Next we consider additional thread sizes that may appear as a result the other speculative parallelization overheads. We call these the *overhead thread sizes* and call the complete table with all thread sizes the *extended thread table*.

3.2 Base Thread Sizes and Intrinsic Load Imbalance Overhead

In this section we discuss how to compute the base thread sizes from the static program code. These appear through possible execution of different control paths. Workload variations across threads are mainly caused by the following factors: conditional statements, inner loops, and cache misses. In practice, the load imbalance overhead will be amplified by combinations of these factors, such as a conditional statement, with an inner loop.

To compute the base thread sizes, we use a variation of the control-flow graph (CFG), which we call a *collapsed control-flow graph (CCFG)*. To build the CCFG we annotate each basic block node in the CFG with the estimated execution time of the instructions in this basic block. When building the CCFG for a particular loop that is also being considered for speculative execution, we collapse all its inner loops and procedures into a subgraph with an estimate of the execution time of the control paths in these inner loops or procedures. In this way, the CCFG of properly structured programs should not have any backward

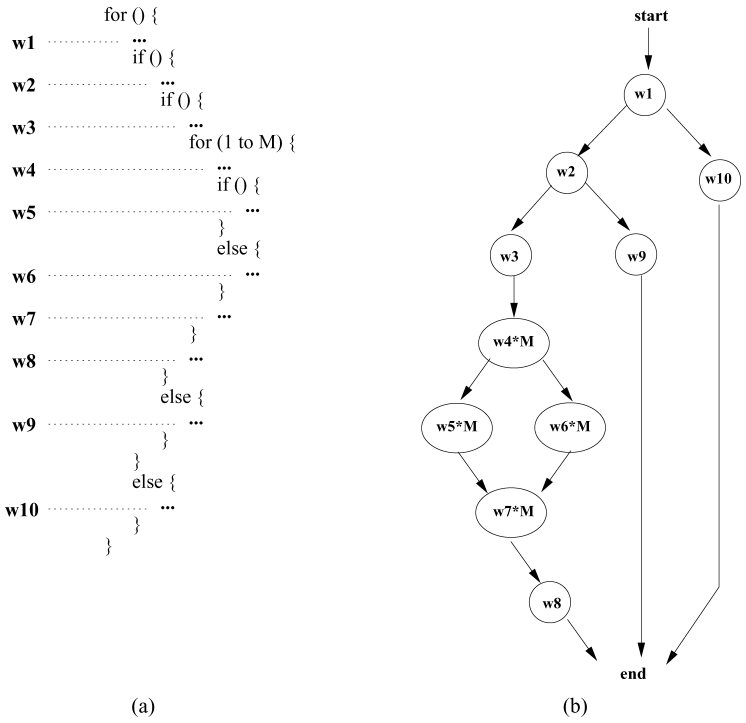

Fig. 2. Example loop being considered for speculative parallel execution (a); and its corresponding collapsed control-flow graph (CCFG) (b). The labels inside the basic blocks correspond to their estimated execution times. In this example, M is the iteration count of the inner loop.

edges and is then a directed acyclic graph. The CCFG is somewhat similar to the hierarchical task graph (HTG) [Girkar and Polychronopoulos 1994], except that it contains no hierarchical information.

As an example, Figure 2a shows the skeleton of a loop in C-like syntax and Figure 2b shows the corresponding CCFG. Note how in the CCFG a loop is represented by an acyclic subgraph where the weights of the arcs are multiplied by the iteration count, M.

For the base thread sizes to be accurate, it is important that the CCFG be built from an intermediate representation, whose basic block code is very close to the final machine code to be produced by the code generator. On the other hand, some speculative parallelization analyses and transformations are likely to be performed using some high-level representation. Thus, ideally, a close collaboration between the code generator and the speculative parallelizer is desirable.

With the CCFG, we can easily generate all the possible execution paths from the start node to the end node, along with their respective estimated execution times and execution probabilities. Execution probabilities can be generated by simply assigning equal probability to each direction of a conditional statement or through some more elaborate static or dynamic

mechanism for estimating the probabilities of the different directions.[1] Our results indicate that the simple equal probabilities heuristic leads to reasonable results.

3.3 The Extended Thread Tuple Model

The tuple model proposed in Dou and Cintra [2004] starts with all possible thread sizes and computes the probabilities associated with all possible tuples. In doing so, that model assumes that the probability that a certain thread size appears in a processor slot in a tuple does not depend on the processor slot. A more accurate model should consider how the probabilities of occurrences of thread sizes vary with the processor slot. For instance, a thread running in processor zero, which we assume is always the nonspeculative thread in a tuple, cannot have a size that corresponds to a thread that overflows the speculative buffer. Similarly, the probability that a thread size that corresponds to a thread that has been squashed appears in a certain processor slot in a tuple actually depends on the processor slot: the probability that this squashed size appears in a more speculative processor slot is greater than the probability that it appears in a less speculative processor slot, since, in the first case, there are more chances that the producer thread will appear in some of its predecessor processor slots. Here the original tuple model is extended to accommodate these variations in probability values.

In the tuple model, the possible thread sizes are divided in two groups: the *base thread sizes*, which are those that correspond to the overhead-free execution of all possible execution paths, and the *overhead thread sizes*, which are those that are generated by adding some overhead to a base thread size. Thus, by definition, every overhead thread size has a corresponding base thread size from which it originates. Section 3.2 showed how to obtain the base thread sizes using the CCFG.

Assume that the possible control-flow paths originate thread sizes in an ordered set B, with M (or $|B|$) entries, with probabilities of occurrences p_1, ..., p_M.[2] Further assume that B joined with the overhead thread sizes forms a new ordered set E, which has N (or $|E|$) possible thread sizes $W_1, \cdots, W_N$. The sets are ordered according to increasing thread sizes. Thus, $E = \{W_1, W_2, \cdots, W_N\}$ and $W_1 \leq W_2 \leq \cdots \leq W_N$. Then, there are N^P ways in which these thread sizes can be combined to form thread tuples. More formally, a thread tuple is an element of E^P, where $E \times E$ is the Cartesian product. The probability of occurrence of a particular thread tuple i, which we refer to by p_{tuple_i}, is the product of the probabilities of occurrences of each thread size in *a particular processor slot* in the tuple. These p_{tuple_i} are then used to compute the overall

[1]Note that the problem of estimating the probability of conditional statements is usually more complex than the problem of estimating the probability of branches, in general. This is because loop-controlling branches are usually more predictable than nonloop branches. This problem is also more complex than the simpler problem of branch prediction, which only involves estimating the most likely direction of a branch.

[2]We implicitly assume that the events associated with the parallel threads following some control path are independent, which may not always be true when there is some correlation among certain paths.

estimated sequential and parallel execution times as:

$$Tseq_{est} = \sum_{tuple_i \in B^P} Tseq_{tuple_i} p_{tuple_i} \quad (1)$$

$$Tpar_{est} = \sum_{tuple_i \in E^P} Tpar_{tuple_i} p_{tuple_i} \quad (2)$$

where $Tseq_{tuple_i}$ and $Tpar_{tuple_i}$ are the sequential and parallel execution times for tuple i. Thus, to compute the overall estimated sequential execution time of the average tuple, we only consider base thread tuples that are intrinsically derived from the possible execution paths and do not suffer from overheads of speculative execution, which is what one would expect from a sequential execution of the code section. For a given thread tuple i, the parallel and sequential execution times for the tuple are given by:

$$Tseq_{tuple_i} = \sum_{W_j \in tuple_i} W_j \quad (3)$$

$$Tpar_{tuple_i} = \max_{W_j \in tuple_i} W_j \quad (4)$$

Equation (3) simply indicates that the sequential execution time of the group of threads in a thread tuple is given by the sum of their execution times, while Eq. (4) indicates that the parallel time of the same group of threads is simply the execution time of the largest of the threads. Note that, in this way, Eq. (4) is an approximation to the execution model of Figure 1c, as it does not take into account the relative position of the largest thread.

A naive computation of Eq. (1) would involve enumerating all possible base tuples and would, thus, have a computational complexity of $O(M^P)$, where M is the number of base threads. However, it can be easily shown that since the assignment of thread sizes to processors in a thread tuple can be seen as independent discrete random variables from the same distribution, Eq. (1) is equivalent to:

$$Tseq_{est} = P \sum_{W_i \in B} W_i p_i \quad (5)$$

which can be computed in $O(M)$.

Similarly, to simplify the computation of Eq. (2), it is converted using Eq. (4) into:

$$Tpar_{est} = \sum_{W_j \in E} W_j \, p(Tpar_{tuple_i} = W_j) \quad (6)$$

where the second term is the probability that the parallel time of a tuple is equal to a given thread size. To derive this term, we define $p_{i,j}$ as the probability that thread size $W_i \in E$ appears in processor j in a given tuple (the processor slots are numbered from 0 to $P - 1$). The computation of these $p_{i,j}$ terms is divided according to the type of thread W_i.

Case 1: W_i is a base thread size and it does not generate any new thread sizes with overheads. In this case, the probabilities are given by:

$$p_{i,j} = p_i \tag{7}$$

where p_i, as mentioned earlier, is simply the probability of execution of the control path that leads to the base thread size W_i.

Case 2: W_i is a thread size related to squash overheads (i.e., either an overhead thread size generated from squash or the corresponding base thread size). In this case, we define W_{prod} as the base thread size that produces the value that causes the violation (and, thus, p_{prod} is its probability), and W_{base} as the base thread size that consumes the value and may suffer the violation. Then, the probabilities of both the squashed and the original, nonsquashed, thread sizes are given by:

$$p_{i,j} = \begin{cases} (1-p_{prod})^j\, p_{base} + (1-(1-p_{prod})^j) p_{base}(1-p_{dep}) \\ \text{, if } W_i \text{ is a base size} \\ (1-(1-p_{prod})^j) p_{base} p_{dep}, \quad \text{if } W_i \text{ is a squashed size} \end{cases} \tag{8}$$

where p_{dep} is the probability that the dependence occurs in an execution instance of the pair W_{prod} and W_{base}. Note that for a base thread size W_i and its squashed counterpart W_k, we have $p_{i,j} + p_{k,j} = p_i$ for every processor j. Note also that $p_{i,0} = 0$ for a squashed thread size W_i.

Case 3: W_i is a thread size related to overflow overheads (i.e., either an overhead thread size generated from overflow or the corresponding base thread size). In this case, we define: W_{base} as the base thread size that may overflow the buffer and become stalled; *firstlonger* as the rank in B of the first thread size whose execution time is greater than the time of occurrence of the store that causes the overflow, and the stall (a thread size W_i that overflows the speculative buffer at time t can only stall because of predecessors of size greater than t); and, for a stalled thread size, *waitfor* as the rank in B of the predecessor thread for which the stalled thread has to wait. Note that for a given overhead thread size W_i, *waitfor*, *base*, and *firstlonger* are unambiguously defined. The probabilities of the overflowed and the original thread sizes are given by:

$$p_{i,j} = \begin{cases} (1-p_{ovflow}) p_{base} + p_{base} p_{ovflow} \left(1 - \sum_{k=firstlonger}^{|B|} p_k\right)^j \\ \text{, if } W_i \text{ is a base size} \\ \left(\left(\sum_{k=1}^{waitfor} p_k\right)^j - \left(\sum_{k=1}^{waitfor-1} p_k\right)^j\right) \cdot p_{ovflow} \cdot p_{base} \\ \text{, if } W_i \text{ is an overflowed size} \end{cases} \tag{9}$$

where p_{ovflow} is the probability that the overflow occurs in an execution instance of W_{base}. Note that for a base thread size W_i and its overflowed counterparts $W_{k_1}, \cdots, W_{k_n}$ (each for a different longest predecessor *firstlonger*), we have $p_{i,j} + p_{k_1,j} + \cdots + p_{k_n,j} = p_i$. Note also that $p_{i,0} = 0$ for an overflowed thread size W_i.

Case 4: W_i is a thread size related to interthread communication overhead (i.e., either an overhead thread size generated from waiting for communication or the corresponding base thread size). In this case, we define W_{prod} as the base thread size that produces the value that is involved in the communication and W_{base} as the base thread size that consumes the value and may have to wait for the communication. When the interthread communication is set up dynamically based on data addresses (Section 3.4.4 and Cintra and Torrellas [2002]; Moshovos et al. [1997]; Steffan et al. [2002]) the communication, and, thus, the stall, will only occur when the dependence does occur. Then, the probabilities of both the stalled and the original, nonstalled, thread sizes are given by Eq. (8).

When the interthread communication is set up statically based on the static memory references (Section 3.4.4 and Krishnan and Torrellas [1999]; Sohi et al. [1995]) the communication will always occur and it is assumed that a suitable producer will always be present to avoid deadlocks. In this case, the probabilities of both the stalled and the original, nonstalled, thread sizes are given by:

$$p_{i,j} = \begin{cases} p_{base} & \text{, if } W_i \text{ is a base size and } j = 0\text{; or } W_i \text{ is a stalled size and } j \neq 0 \\ 0 & \text{, if } W_i \text{ is a base size and } j \neq 0\text{; or } W_i \text{ is a stalled size and } j = 0 \end{cases} \tag{10}$$

Note that in Eq. (8) and (10) for a base thread size W_i and its stalled counterpart W_k, we have $p_{i,j} + p_{k,j} = p_i$ for every processor j. Note also that $p_{i,0} = 0$ for a stalled thread size W_i in both equations.

With the values of $p_{i,j}$ computed as described above, we can now compute $p(Tpar_{tuple_i} = W_j)$, the probability that the parallel time of a tuple i is equal to some thread size $W_j \in E$. This term can be computed as (recall that the thread sizes are sorted in increasing order):

$$p(Tpar_{tuple_i} = W_j) = \prod_{k=0}^{P-1} \left(\sum_{l=1}^{rank(j)} p_{rank(l),k} \right) - \sum_{m=1}^{rank(j)-1} p(Tpar_{tuple_{i'}} = W_{rank(m)}) \tag{11}$$

where $rank(x)$ is a function that maps the indexes used to name threads, e.g., i in W_i, to the rank of each thread in the ordered set E.[3] Note that to compute the estimated average parallel execution time, we consider all N thread sizes, including those that arise due to speculative execution overheads. The first term in Eq. (11) is simply the probability that the parallel time of a tuple is equal to any of the thread sizes up to $rank(j)$, inclusive. The second term in Eq. (11) is then the probability that the parallel time of a tuple is equal to any of the thread sizes up to $rank(j-1)$. Note that Eq. (11) can be computed recursively and there is no need to recompute the second term for each j. In the notation for summations, we assume that when the upper bound is less than the lower bound, the sum defaults to a value of zero. Thus, the second term defaults to zero when $j = 1$.

[3]This naming indirection is necessary, because we name overhead threads after all base threads have been named and ordered in set B.

With the results of Eq. (11) we can then compute Eq. (6), and, finally, with this result and that of Eq. (5), we can compute the estimated speedup as:

$$S_{est} = \frac{Tseq_{est}}{Tpar_{est}} \tag{12}$$

The Appendix contains an example of how to use the above equations; Section 3.4 elaborates on how to generate the overhead thread sizes required by the model. Finally, note that when the number of loop iterations is known to be less than the number of processors in the system, the value of P in the formulas above will be replaced with the number of iterations.

3.4 Overhead Thread Sizes

The previous section introduced the extended thread tuple model and assumed that all thread sizes, base and overhead, had been created and ordered in the B and E sets. The computations of the previous section also assumed that the probabilities of occurrence of some overheads, such as p_{dep} in Eq. (8) and p_{ovflow} in Eq. (9), had also been predetermined. This section explains in detail how the thread sizes with overheads are created, starting from the base thread sizes (whose generation is explained in Section 3.2).

3.4.1 *Initial Considerations.* The starting point of the tuple model is to compute the base thread sizes and their probabilities of occurrence. As an example, Figure 3a and b show a case where there are two possible base thread sizes. Next, the model adds the overhead thread sizes and their probabilities. As the overhead thread sizes are derived from the original base thread sizes, the original probabilities of occurrence of the base threads are divided into two separate probabilities: the overhead thread's probabilities of occurrence and the base, nonoverhead, thread size's probabilities of occurrence. The sum of these two terms must be the original probability of execution of the base thread size.

To simplify the problem, and to comply to the mathematical model of Section 3.3, we impose the restriction that a base thread size can only possibly suffer from one source of overhead. Thus, a base thread size cannot be squashed and, at the same time, overflow the speculative buffer and stall. The only exception is dispatch and commit overheads, which can be added to all thread sizes, including overhead ones, as described in Section 3.4.5. Another restriction is that an extended thread size cannot generate any further thread sizes by either squashing threads or by forcing stalled successors to wait. Note that these restrictions imply that $N < M^2$, where N is the total number of thread sizes and M is the number of base thread sizes. In case some base thread size may suffer from more than one source of overhead, then one overhead must be chosen. For this purpose, we propose choosing the overhead that would appear first: e.g., if the squashing store in the producer predecessor is expected to appear before the stalling store in the thread itself, then consider that the thread is only squashed, and vice versa.

3.4.2 *Squash and Restart Overhead.* Since the major contribution in the squash and restart overhead is the reexecution of part of the thread after being

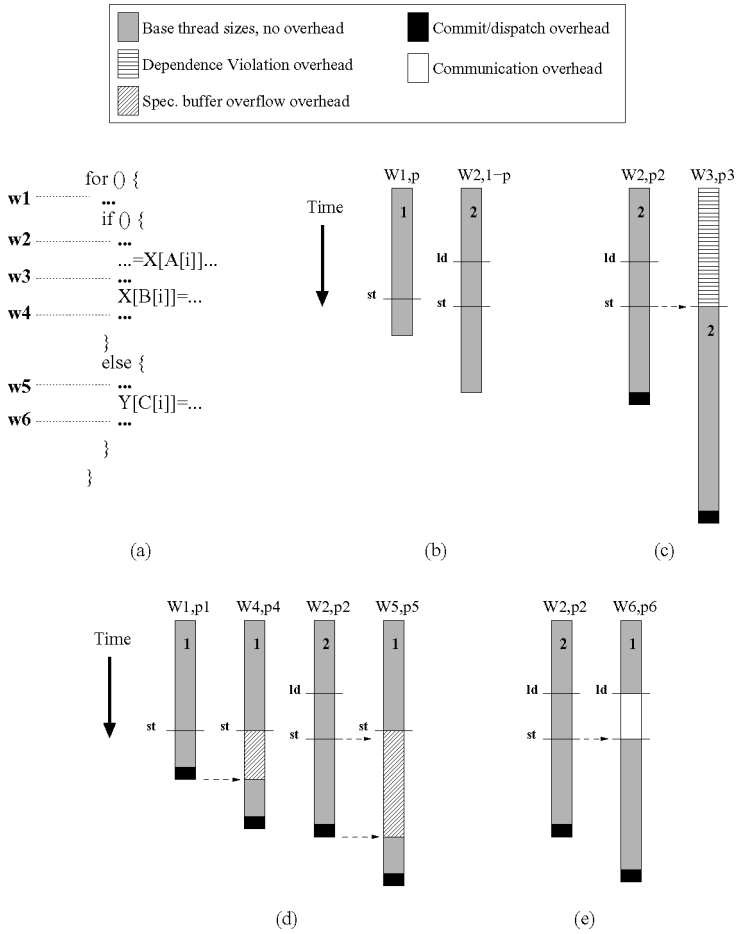

Fig. 3. Example of modeling speculative parallelization overheads in the extended tuple model: skeleton code to be speculatively parallelized (a); base thread sizes and probabilities from the two possible execution paths (b); additional thread sizes and probabilities when overheads are included (c), (d), and (e). The numbers inside the bars identify the base thread sizes.

squashed, the final size of the squashed thread is the original thread size plus the execution time of the predecessor producer thread until the violation is detected, which is usually shortly after the store involved in the RAW violation. Figure 3c shows the case of a potential data dependence violation and squash, in which case the original base thread size $W2$ originates a new thread size $W3$.

The probabilities of occurrence of the squashed thread size and of the original base thread size are computed through Eq. (8). The rationale behind this equation is the following. In order for a squashed thread size to appear in a given processor slot it is necessary that three conditions be true: (1) the base thread appears in this processor slot; (2) the producer thread is present in at least one of its predecessor processor slots; and (3) the data dependence does occur in this particular execution instance of the producer and consumer threads. The

probability of the squashed thread is then the joint probability of these events (which are considered to be independent). The term $(1-(1-p_{prod})^j)$ in Eq. (8) is the probability that a producer thread is present in at least one of its predecessor processor slots. The term p_{base} is the probability that the base thread appears in a particular processor slot. Finally, the term p_{dep} is the probability that the data dependence does occur. Note that the result of Eq. (8) is zero for a squashed thread size and processor zero, which is to be expected since no squashed thread size should appear on the nonspeculative processor.

In order for the original base, nonsquashed, thread size to appear in a given processor slot, it is necessary that the base thread appears in this processor slot and that one of two conditions occur: (1) the producer thread is not present in any predecessor processor slot or (2) if that is not the case, the data dependence does not occur. The term $(1-p_{prod})^j$ in Eq. (8) is the probability that no producer thread appears in any predecessor processor slot and corresponds to the first condition. The term $(1-(1-p_{prod})^j)$ is the probability that a producer thread is present in at least one of its predecessor processor slots and the term $(1-p_{dep})$ is the probability that the data dependence does not occur. Thus, combined, these two terms correspond to the second combination of conditions. Note that the result of Eq. (8) is simply p_{base} for a nonsquashed thread size and processor zero, which is to be expected, since only the nonsquashed size should appear on a nonspeculative processor.

The methodology described above to model squash and restart overheads does not take into account the fact that at a violation not only the consumer thread but also all its successors must be squashed. This inaccuracy is likely to have more significant impact in systems with large number of processors. We decide to trade off accuracy for simplicity of the model.

Estimating the probability of data-dependence violations under speculative parallelization (i.e., p_{dep}) is difficult, because, by construction, these are highly unpredictable (more predictable data dependences are more efficiently handled by explicit synchronization [Cintra and Torrellas 2002; Moshovos et al. 1997; Steffan et al. 2002]). Estimating the likelihood of violations can be done with static probabilistic memory disambiguation analysis [Chen et al. 2003; Zhai et al. 2004] or profiling [Liu et al. 2005; Marcuello and González 2002; Olukotun et al. 1999].

3.4.3 *Speculative Buffer Overflow Overhead.* The major contribution in the speculative buffer overflow overhead is the idle time the thread spends waiting for all its predecessor threads to commit, which is approximately the idle time waiting for its longest predecessor to complete execution. Thus, the size of a thread that stalls because of speculative buffer overflow is equal to the execution time of the predecessor thread plus the execution time remaining to complete execution once the thread is released. In practice, several thread sizes could assume this role of longest predecessor, as long as their execution time is longer than the timestamp of the stalling store. To keep the model simple we only consider predecessor thread sizes that are base thread sizes. Thus, in the model, from the original thread size (the one suffering the speculative buffer overflow),

we generate as many new thread sizes as there are base thread sizes with execution times longer than the timestamp of the stalling store (note that this includes the original base thread size itself). Figure 3d shows the case of a potential speculative buffer overflow of thread size $W1$, in which case it must wait for either another instance of $W1$ or an instance of $W2$ generating $W4$ or $W5$, respectively.

The probabilities of occurrence of an overflowed thread size and of the original base thread size are computed through Eq. (9). The rationale behind this equation is the following. In order for an overflowed thread size, corresponding to a certain stalling predecessor thread, to appear in a given processor slot it is necessary that four conditions be true: (1) the base thread appears in this processor slot; (2) the stalling thread is present in at least one of its predecessor processor slots; (3) no thread longer than the stalling thread appears in any predecessor processor slot; and (4) the overflow does occur in this particular execution instance of the base thread. The probability of an overflowed thread is then the joint probability of these events (which are assumed to be independent). The term $(\sum_{k=1}^{waitfor} p_k)^j$ in Eq. (9) is the probability that only base thread sizes in the range $k = 1$ to $k = waitfor$ appear in all of its predecessor processor slots. Similarly, the term $(\sum_{k=1}^{waitfor-1} p_k)^j$ is the probability that only base thread sizes in the range $k = 1$ to $k = waitfor - 1$ appear in all of its predecessor processor slots. Thus, the difference of the two terms corresponds to conditions (2) and (3). The term p_{base} is the probability that the base thread appears in a particular processor slot. Finally, the term p_{ovflow} is the probability that the overflow does occur. Note that the result of Eq. (9) is zero for an overflowed thread size and processor zero, which is to be expected since the nonspeculative processor will never stall.

In order for the original base, nonoverflowed, thread size to appear in a given processor slot, it is necessary that the base thread appears in this processor slot and that one of two conditions occur: (1) the overflow does not occur or (2) if that is not the case, no thread size with running time larger than the timestamp of the stalling store appears in any of the predecessor processor slots. The term p_{base} in Eq. (9) is the probability that the base thread appears in a particular processor slot. The term $1 - p_{ovflow}$ is the probability that the overflow does not occur and corresponds to the first condition. The term $(1 - \sum_{k=firstlonger}^{|B|} p_k)^j$ is the probability that no thread size between $k = firstlonger$ to $k = |B|$ appears in any of the predecessor processor slots and corresponds to the second condition. Note that the result of Eq. (9) is simply p_{base} for a nonoverflowed thread size and processor zero, which is to be expected, since only the nonoverflowed size should appear on a nonspeculative processor.

As a new overflowed thread size is generated for each base thread size $W_{waitfor}$ in the range $firstlonger \leq waitfor \leq |B|$, after generating all the overflowed thread sizes, the worst case number of the overflowed thread sizes is $|B|^2$, when base thread size overflows with $firstlonger = 1$.

Determining the probability of a speculative buffer overflow (i.e., p_{ovflow}) is also a very difficult problem. In systems that have dedicated fully-associative speculative buffers, as in Hammond et al. [1998], this problem reverts to the

problem of counting the number of speculative buffer lines covered by the stores in the execution path under consideration. In systems that have only the set-associative L1 caches, as in Sohi et al. [1995], the problem is that of finding whether two memory accesses conflict in the cache. This problem has been well addressed for regular applications (e.g., Chatterjee et al. [2001]; Vera and Xue [2002]), but is still an open problem for irregular applications.

3.4.4 *Interthread Communication Overhead.* Some systems support, in addition to data speculation, explicit synchronized communication through memory or registers [Cintra and Torrellas 2002; Krishnan and Torrellas 1999; Moshovos et al. 1997; Sohi et al. 1995; Steffan et al. 2002]. This can be modeled as additional execution time added to the base thread between the time of the consumption and the production of the data. Thus, the size of a thread that stalls because of synchronized interthread communication is its original execution time plus the time between the stalling load and the releasing store. Figure 3e shows the case of a potential synchronization and interthread communication, in which case the potential memory dependence on array X is handled by explicit synchronization and the original base thread size $W2$ originates a new thread size $W6$.

Two approaches for synchronized interthread communication have been proposed. In one approach the synchronization is placed dynamically at runtime [Cintra and Torrellas 2002; Moshovos et al. 1997; Steffan et al. 2002] and is usually triggered only when a dependence does occur, or is thought to occur. In this case, the probabilities of occurrence of the synchronized thread size and of the original base thread size are computed through Eq. (8). The rationale behind this equation is analogous to that explained in Section 3.4.2 and is not repeated here for the sake of brevity.

In the second approach, the synchronization is placed statically [Krishnan and Torrellas 1999; Sohi et al. 1995] and is usually triggered each time the load appears. In this case, the probabilities of occurrence of the synchronized and of the original base thread size are computed through Eq. (10). The rationale behind this equation is the following. The synchronized thread size cannot appear in the nonspeculative processor slot and will appear each time the original base thread size would have appeared in any other processor slot. The original thread size, on the other hand, cannot appear in any speculative processor slot and will not be replaced when it appears in the nonspeculative processor slot. Thus, the probability that the original base thread size (synchronized thread size) appears in processor zero is p_{base} (zero) and in the other processor slots is zero (p_{base}).

3.4.5 *Dispatch and Commit Overheads.* Different from the other overheads, thread dispatch and commit overhead is not represented as additional thread sizes in the model. Instead, the time required by these operations is added to the execution time of the existing thread sizes. However, because of the in-order commit requirement, the amount of dispatch and commit overhead observed by a thread is not exactly the nominal time required by these operations, but depends on the position of the longest thread in the tuple. Simply adding the nominal cost of the operations to the thread sizes would incorrectly overlap some

of the commit costs. In reality, when the longest thread is the nonspeculative one, the total overhead is equal to P times the nominal dispatch and commit time, and when the longest thread is the mostspeculative one, the total overhead is simply equal to the nominal dispatch and commit time. In the, average case, the total overhead is equal to $T_{disp/comm}(\sum_{j=1}^{P} j)/P = T_{disp/comm}(P+1)/2$, where $T_{disp/comm}$ is the nominal dispatch and commit time, which we assume constant for simplicity. Figure 3 shows the commit and dispatch overheads added to all the thread sizes.

The thread dispatch time is usually a somewhat fixed time required to update some system data structures, possibly copy-in some register values, and dispatch the new thread on the processor. The commit time, however, is more variable as it depends on the amount of data that is modified by the thread, while it is speculative. The analysis required to compute this is very similar to that required to estimate speculative buffer overflows. This overhead also depends on the contention at the bus during the write-backs of data from the speculative buffers. Some speculative multithreaded systems support "lazy commit," whereby data modified by a nonspeculative thread may reside in the speculative buffer after the processor starts work on a more speculative thread and is lazily written back to memory on demand [Garzarán et al. 2003]. This, however, requires additional costly hardware.

3.5 Complexity Analysis

The computational complexity of the proposed framework is composed of two parts: the generation of all thread sizes and the computation of the estimated speedup. The first is mainly a function of the number of control paths in the program (M) and the number of memory references tracked for all base thread sizes (R). The latter is mainly a function of the number of base thread sizes (M) and the number of processors (P).

The complexity analysis of generating all thread sizes is as follows. From the CCFG representation it is possible to generate all base thread sizes in $O(M)$, if information is carried and stored at intermediate nodes to avoid traversing any path segment more than once. Searching for possible speculative buffer overflow occurrences requires considering all memory references for each base thread, which leads to $O(R)$. Afterward, generating the stalled thread sizes because of overflow requires comparing the timestamps of overflowing stores against all base thread sizes, which leads to $O(M^2)$. Searching for possible data-dependence violations would require comparing the target and timestamps of each memory reference in a base thread against all memory references in all other threads. By storing all the memory references in a hash table, it is possible to reduce the *expected* complexity to $O(R)$ (with $O(R^2)$ worst-case). Thus, the overall expected complexity of generating all thread sizes is $O(M^2 + R)$.

The complexity analysis of computing the estimated speedup, given the thread sizes, is as follows. The complexity of Eq. (5) is $O(M)$. The complexity of Eq. (6) is determined by the complexity of computing the $p(Tpar_{tuple_i} = W_j)$ terms, which, using Eq. (11), is $O(NP)$ once all $p_{i,j}$ terms are computed. The computation of each $p_{i,j}$ term through Eqs. (7), (8), and (10) is $O(1)$, but is $O(M)$

through Eq. (9). Since there are NP $p_{i,j}$ terms, the complexity of generating all such terms is $O(NMP)$. Thus, the overall complexity of Eq. (11), and, thus, Eq. (6), is $O(NMP)$. Finally, the complexity of the computation of the estimated speedup through Eq. (12) is then also $O(NMP)$. Recall that, as explained in Section 3.4.1, $N = O(M^2)$ in the worst case. Finally, the overall complexity of the framework is then $O(M^2 + R + NMP)$.

3.6 Limitations and Optimizations

To keep the complexity of the framework tolerable, we were forced to make some simplifications. These simplifications can be classified into two categories: model simplifications and implementation simplifications. The first are those that aim to reduce the complexity of the cost model and are, thus, intrinsic to the model itself. Eliminating such simplifications would require reworking the model. The latter are those that were required in order to keep the implementation of the compiler framework prototype manageable. These are not intrinsic to the model itself and can be reduced, or eliminated, with the incorporation of more elaborate static and/or profile analyses to the compiler framework. Many of the model limitations were already discussed alongside the presentation of the model. Next, we discuss some other limitations, along with some optimizations.

3.6.1 *Limitations of the CCFG and Optimizations.* As described, the CCFG does not correctly represent the execution paths and execution times in the presence of inner loops. An inner loop with M iterations and with conditional structures leading to N, possible control paths can generate up to N^M combinations of paths at runtime. Instead of trying to represent these (prohibitively) many paths in the CCFG, we choose to represent only a total of N possible paths that are executed M times (Figure 2). This corresponds to the case where all iterations in the inner loop execute the same path in a given execution of the inner loop (different execution instances of the inner loop can follow different paths). It is clear, then, that some possible execution paths (and, thus, thread sizes) would not be considered by the cost model. It can be shown that the range of thread sizes is the same in both cases, despite the model having fewer sizes. For this reason, we expect the model to predict more load imbalance, which is likely to result in an underprediction of the speedup.

Another weakness of the CCFG is that it cannot easily handle recursive procedure calls. In the presence of recursion, we choose to represent in the CCFG only a limited number of the recursive calls and decide whether to build the procedure's subgraph depending on the probability of occurrence of the path leading to the recursive call. If the probability is below a threshold, then we simply ignore the procedure call and assign it a null workload.

As an optimization of our CCFG representation, we ignore thread sizes on paths leading directly to `exit` and `break` statements. This is reasonable, since these paths lead to the termination of the speculative execution and, thus, should not be included in the thread size mix.

3.6.2 *Limitations of the Model and Optimizations.* One limitation of the model is that we decide not to consider cascaded thread interactions. For

instance, a thread may stall because of a speculative buffer overflow before it reaches the store that causes a data-dependence violation with a successor thread; or a thread may stall because of a speculative buffer overflow and then wait for a predecessor that has also stalled as a result of a speculative buffer overflow. To model these, we would have to apply the overhead models not only to base thread sizes, but also to thread sizes that already incorporate some overheads. In practice, this would make the model significantly more complex. Note also that there is the possibility of a combination on the same thread of a data-dependence violation followed by a speculative buffer overflow when the thread is reexecuted. There is no other possible combination except with dispatch and commit overhead.[4] Again, we decide not to model this compound overhead to keep the model simple.

Another limitation of the model is that it focuses on estimating sources of overheads and assumes that all performance gains come from the parallel execution of threads. However, it has been observed that, in some cases, a significant fraction of the performance gains from speculative parallelization come from a prefetching effect to lower level shared caches. In this case, despite the speculative threads not being able to commit a significant amount of work, they still help the nonspeculative thread by accessing data that it will need later. Incorporating this source of speedup in the model is left as future work.

Another limitation is also, that the compiler model attempts to emulate the simple execution model described in Section 2.1. Some hardware architectures support a more flexible execution model by way of more complex hardware [Colohan et al. 2006; Garzarán et al. 2003; Renau et al. 2005]. Such support, however, is not yet well established and the complexity versus performance tradeoffs are still being explored. Thus, extending the compiler model to handle such execution models is left as future work.

Section 3.5 argued that the formulas used by the model have only a polynomial complexity dependence on the number of thread sizes. Nevertheless, when the speculative section has several possible execution paths and many potential overheads, the model we propose may result in too many thread sizes. If compilation time becomes an issue, then some general optimizations may be applied to reduce the number of thread sizes while introducing small errors. One possible optimization is to group threads whose sizes differ by only a small percentage. The resulting thread size could be as long as the average of the individual thread sizes and the resulting probability of occurrence would be the sum of the probabilities of the thread sizes. Another optimization is to ignore some thread sizes whose probabilities of occurrence are too small and whose sizes are within a certain fraction of the remaining thread sizes. Such threads would have a small impact on the overall speedup.

3.6.3 *Limitations of the Prototype Implementation.* A significant problem of all static compiler models, such as ours, is the fact that many parameters to

[4]In case the squashed thread has already finished and is waiting to commit by the time the violation is detected, the idle time is attributed to squash overhead and not load imbalance overhead. Also, in case a thread stalls because of overflow and is squashed before it can resume, the idle time is attributed to squash overhead and not overflow overhead.

the model may only be available at runtime. For example, the model requires the probability of occurrence of the two directions in conditional statements to compute the probability of each thread size. Another example of an important parameter to the model is the iteration count of inner loops. While there exist a few static techniques that may help (e.g., Patterson [1995]; Wagner et al. [1994]), implementing those would significantly digress from the focus of our work. Nevertheless, we note that integrating such static analyses with our model is straightforward, as the result of the analyses could be easily added to the CCFG. An alternative to such static analyses is profiling. Again, we note that such techniques could be easily integrated with our model, but doing so is beyond the scope of this paper.

4. EXPERIMENTAL SETUP

4.1 Compilation Infrastructure

We implemented the algorithms and formulas described in Section 3 using the SUIF1 compiler infrastructure [Hall et al. 1996]. All the compiler analyses are done at the high-level intermediate representation (IR) of SUIF. Our new pass was added to the end of the *pscc* chain.

In this work we only consider the tuple model and the heuristics for estimating load imbalance and squash and restart overheads. Thus, we do not yet add buffer overflow overheads to the model, as described in Section 3.4.3. Dependences are assumed to be true (i.e., $p_{dep} = 1$) whenever threads reading and writing the same program symbol are executed concurrently, so that the overall probability of violations relates directly to the probabilities of execution of paths as in Chen et al. [2003]. For the dispatch and commit overhead, we use the methodology described in Section 3.4.5, but we assume that the nominal commit time is a fixed time, required to write back all the entries in the speculative buffer with some hardware support [Hammond et al. 1998] and that the dispatch overhead is also a fixed time, as suggested in previous work [Hammond et al. 1998; Steffan and Mowry 1998]. Based on the results of Wagner et al. [1994], loops with unknown iteration counts are assumed to have five iterations.

4.2 Applications and Loop Selection

To evaluate our scheme, we use a subset of the SPEC2000 benchmarks, comprising of four floating-point and five integer applications.[5] These applications are representative of the workloads typical for current and future CMP's. To keep the execution time in our simulation environment (Section 4.3) manageable, we use reduced input sets [KleinOsowski and Lilja 2002] and we simulate only 500 million instructions after discarding the initial 100 million instructions.

To focus on the tuple model and the heuristics for estimating load imbalance and squash and restart overheads, we do not consider loops that overflow the speculative buffer at runtime. In addition, to isolate the tuple model and

[5]The benchmarks not included are those incompatible with the SUIF1 front end pass, *snoot*.

Table I. Characteristics of the Applications Studied

Application	Benchmark Type	Number of Loops		% of Seq. Time	
		Without Dep.	With Dep.	Spec	Doall
177.mesa	Floating point	7	2	93	0
179.art	Floating point	10 (5)	8 (6)	99	≈ 0
183.equake	Floating point	28 (18)	24 (16)	74	26
188.ammp	Floating point	4	1	50	0
175.vpr	Integer	4	5 (4)	96	≈ 0
181.mcf	Integer	4	3	40	0
186.crafty	Integer	16	16	23	0
255.vortex	Integer	4	0	< 1	0
256.bzip2	Integer	24 (17)	37 (29)	91	5

load imbalance overhead from the heuristics for squash and restart overhead, we separate the loops into two groups: those that we identify, by manual inspection, as not having dependences and those having dependences. Note that loops in the latter group may or may not suffer data-dependence violations at runtime resulting from the actual timing of speculative loads and stores and the scheduling of threads. The loops in the first group correspond to those used in Dou and Cintra [2004]. To estimate the accuracy of the model, we consider for speculative parallelization all loops except those that are parallelizable by SUIF1 alone (we call these *Doall* loops). When reporting speedups, we also do not consider for speculative parallelization loops that are inside Doall loops or are inside loops chosen for speculative parallelization (this is because our architecture does not support nested parallelism). Table I lists the applications we use along with the number of loops we consider for speculative execution (the numbers in parenthesis are those discounting inner loops), the fraction of the sequential execution time that is taken by these, and by Doall loops.

4.3 Simulation Environment

Several architectures for speculative execution in CMPlike systems have been proposed (e.g. Akkary and Driscoll [1998]; Hammond et al. [1998]; Krishnan and Torrellas [1999]; Marcuello et al. [1998]; Sohi et al. [1995]; Steffan and Mowry [1998]; Tsai et al. [1999]). In this paper, we assume a CMP along the lines of the Stanford Hydra CMP Hammond et al. [1998, 2000]. This system consists of four single-issue processors, each with a private L1 data cache, and a shared on-chip L2 cache with separate read and write buses. Each processor has also a private fully associative speculative buffer. The bus protocol supports both cache coherence and speculative parallelization. Data-dependence violations are handled with squashes and there is no support for direct register communication. Table II shows the configuration parameters of the system we model, which are similar to those listed in a recent publication on the Hydra CMP [Prabhu and Olukotun 2003].

To measure the sequential execution times, we run the applications on a x86 simulator based on Virtutech's Simics [Magnusson et al. 2002]. During these simulations, we collect traces with the start and end times of all the loop iterations. These traces also include all the memory addresses with timestamps

Table II. Parameters of the Speculative CMP Modeled

Processor Param.	Value
Number of processors	4
Issue width	1
Memory Param.	Value
L1, L2 size	16KB, 2MB
L1, L2 assoc.	4-way, 4-way
L1, L2 line size	32B, 64B
L1, L2 latency	1, 5 cycles
Main memory latency	50 cycles
Spec. buffer size, assoc.	2 KB, full
Spec. buffer latency	1 cycle

relative to the beginning of the iteration. The total sequential execution time of the speculative loops is then easily computed. Before simulating the parallel execution, we manually eliminate from the traces any cross-iteration communication through registers, since our speculative CMP does not support direct register communication. A nominal dispatch and commit time of 12 processor cycles is assumed. The traces are then fed into a trace-driven simulator that simulates the parallel execution of these threads in a speculative CMP. This simulator properly takes into account the scheduling restrictions of the speculative CMP and performs squashes based on the timestamps of conflicting memory operations. For the loops that are parallelizable with SUIF1, we simply compute their sequential execution time and assume perfect speedup to compute their parallel time. We then report speedups for the execution of all 500 million instructions, including the Doall loops. Note that these loops only contribute to a significant fraction of the speedup in the cases of *183.equake* (Table I).

5. EVALUATION

5.1 Speculative Parallelization Performance

To assess how often speculative parallelization leads to speedups or slowdowns with respect to sequential execution, Figure 4 shows, for each application, a histogram of the actual speedups obtained with the simulations for four processors. From this figure we observe that speculative parallelization leads to a broad range of speedups and slowdowns: on average[6] 56% of the loops lead to slowdowns, while the other 44% lead to speedups. These results show the importance of being able to selectively apply speculative parallelization.

5.2 Model Accuracy

5.2.1 *Outcome Prediction Accuracy.* The output of our compiler model is an estimate of the actual value of the speedup. The primary use of this prediction is to switch off speculative parallelization of loops that are expected to lead to slowdowns and speculatively parallelize those that are expected to

[6]This average is computed with the *number* of loops for each application as the weighting factor.

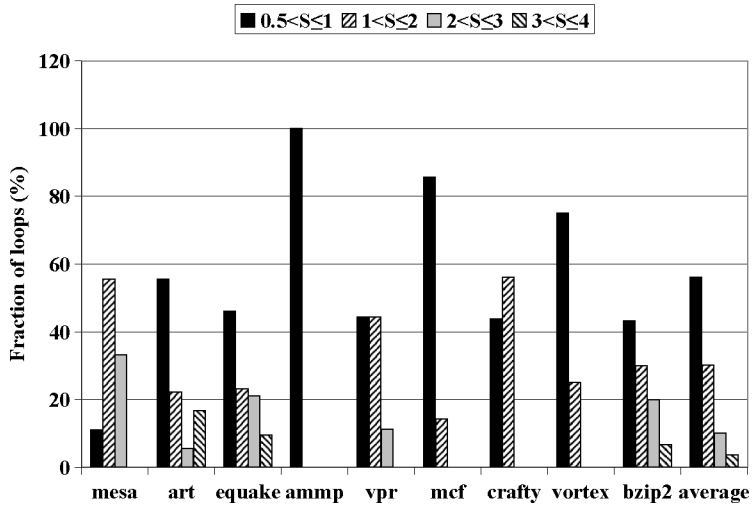

Fig. 4. Histogram of speedups for the speculative loops. A speedup (S) less than one is a slowdown. All loops except Doall are considered.

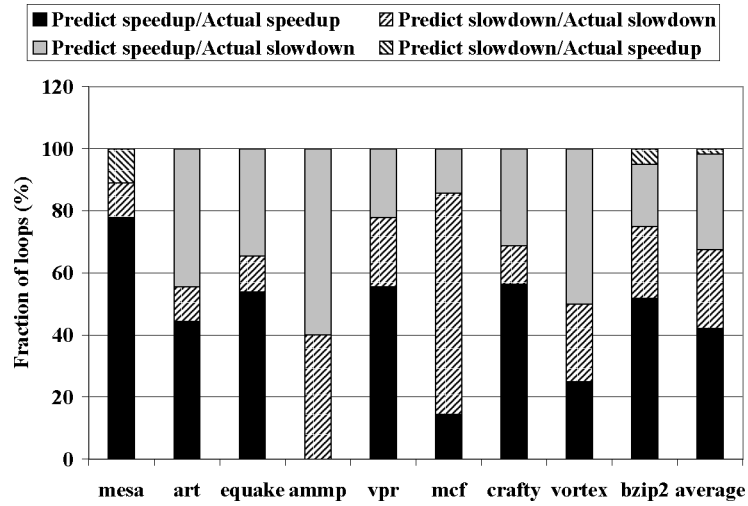

Fig. 5. Outcome of speedup predictions with respect to actual observed speedup or slowdown. All loops except Doall are considered.

lead to speedups. Figure 5 shows, for each application, a breakdown of the fraction of loops that are correctly predicted as leading to speedups (*Predict speedup/Actual speedup*), correctly predicted as leading to slowdowns (*Predict slowdown/Actual slowdown*), incorrectly predicted as leading to speedups (*Predict speedup/Actual slowdown*), and incorrectly predicted as leading to slowdowns (*Predict slowdown/Actual speedup*).

From this figure we can see that our model is accurate enough to correctly identify the speedup or slowdown outcome of the speculative execution in 67% of the cases, on average. Also, it seems that when the model is incorrect, the

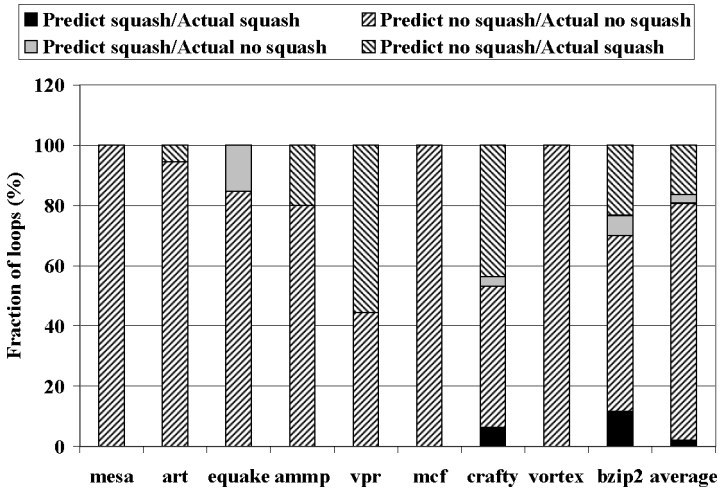

Fig. 6. Outcome of squash predictions with respect to actual observed squash or no squash. All loops except Doall are considered.

tendency is to overestimate the performance gains. This means that the model rarely misses a valid opportunity to speculatively parallelize loops, but that it does not prevent all of the performance degradation from slowdown cases.

5.2.2 *Dependence Violation Prediction Accuracy.* When considering squash and restart overheads, an important intermediate result of the compiler prediction model is its prediction of the occurrence or not of data-dependence violations and, thus, squashes. Figure 6 shows, for each application, a breakdown of the fraction of loops that are correctly predicted as suffering violations (*Predict squash*/*Actual squash*), correctly predicted as not suffering violations (*Predict no squash*/*Actual no squash*), incorrectly predicted as suffering violations (*Predict squash*/*Actual no squash*), and incorrectly predicted as not suffering violations (*Predict no squash*/*Actual squash*). The third case, *Predict squash*/*Actual no squash*, occurs for two reasons: first, our compiler infrastructure does not yet take the indexes of arrays into consideration; and, second, true data dependences may be hidden at runtime if the statically computed timestamps are incorrect and the load actually appears after the store. The fourth case, *Predict no squash*/*Actual squash*, occurs only because our compiler infrastructure does not yet consider violations on scalar variables. Note that we consider that a loop suffers violations as long as a violation occurs in at least one of the invocations of the loop.

From this figure we can see that the model is very accurate in identifying when a loop will not suffer data-dependence violations. On average, 79% of the loops are correctly predicted as not suffering violations and, of those that do not suffer violations, 97% are correctly predicted. Many of such correct predictions are for loops that do have data dependences, but that do not turn into violations, which indicates that the use of (static) timestamps in the model is enough to correctly model the runtime events. However, from this figure we can see that

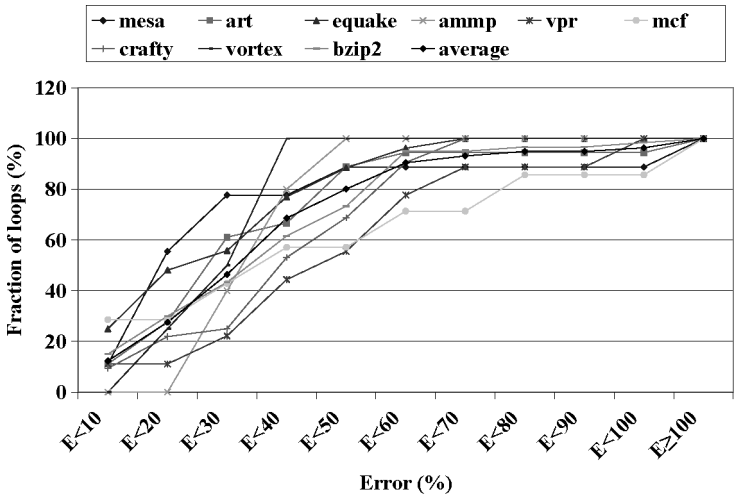

Fig. 7. Cumulative error distributions. All loops except Doall are considered.

the model fails to detect a significant fraction of data-dependence violations. On average, 16% of the loops are incorrectly predicted as not suffering violations and, of those that do suffer violations, only 11% are correctly identified. This is mainly because the current implementation of the model does not take scalar variables into account when looking for possible data dependences.

5.2.3 *Prediction Errors.* In some cases, it may be useful to have an accurate prediction of the actual value of the speedup or slowdown. Figure 7 shows the cumulative error distribution for each application plotted at 10% boundaries. The errors are computed as the absolute difference between the estimated speedups and the actual speedups obtained with the simulation, divided by the actual speedup. Thus, in this figure, a point (x, y) in the curve means that y% of the loops have errors less than x%.

From this figure we can see that our model is reasonably accurate and is able to predict the actual speedup or slowdown within 20% for 28% of the loops, on average,[7] and within 50% for 80% of the loops on average.

A closer look at the results shows that the largest errors appear when the model estimates a smaller speedup (or larger slowdown) than what is actually observed. These cases tend to relate to relatively small loops and the reason for this overestimation of the slowdowns can be attributed to differences between the thread sizes estimated from the SUIF IR and the actual thread sizes. Table III lists the sources of errors for the top 10% loops with the largest errors (19 loops in total), along with the number of times these sources created such large errors.

Finally, while a detailed quantification of the errors induced by the limitations described in Section 3.6 are beyond the scope of this paper, we attempt to provide some quantitative insight into their possible impact.

[7]This average is computed with the *number* of loops for each application as the weighting factor.

Table III. Observed sources of prediction errors for the top 10% of loops with the largest errors[a]

Description of Source	Number of Instances	Range of Errors (%)
Inaccurate thread size estimation with SUIF	8	57 to 137
Biased conditional	5	58 to 67
Unknown iteration count of inner loop	4	77 to 390
Misprediction of the squash overhead	2	58 to 62

[a](One instance under the category of *misprediction of the squash overhead* is also present in the category of *biased conditional*.)

Considering the unavailability of control-flow probabilities: Out of the 190 loops studied, 57% contain one or more conditional statements. Of these, 30% have error less than 20% and 80% have error less than 50%. In contrast, 43% of the 190 loops contain no control-flow statements and, of these, 34% have error less than 20% and 91% have error less than 50%. Thus, overall there seems to be some degradation in accuracy when conditional statements are present, but this does not seem to be a significant source of errors across the loop suite.

Considering the unavailability of inner loop iteration count: Out of the 190 loops studied, 18% contain inner loops with statically unknown inner loop iteration counts. Of these, 11% have error less than 20% and 62% have error less than 50%. In contrast, 3% of the 190 loops contain inner loops with statically known iteration counts. Of these, 20% have error less than 20% and 80% have error less than 50%. Again, there seems to be some degradation in accuracy when inner loops with unknown iteration counts are present, but this does not seem to be a significant source of errors across the loop suite.

5.2.4 *Comparison with Original Tuple Model* [*Do and Cintra* 2004]. Comparing the results with the extended tuple model and including loops that may suffer data-dependence violations against the results with the original tuple model and not including such loops, we can see that prediction accuracy has decreased slightly, but not significantly. The errors reported in Section 5.2.3 are much larger than the ones with the original model, but are mainly because of the larger number of outer loops that have inner loops with statically unknown iteration counts. The main effect of this reduction in accuracy is the increase in the fraction of incorrectly predicted speedups at the expense of correctly predicted speedups; there is little change in the fractions of correctly and incorrectly predicted slowdowns.

5.3 Performance Improvements

Finally, we estimate the actual performance impact of using our model to speculatively parallelize only loops that are predicted to have speedups. Figure 8 shows the overall speedups obtained for the execution of all the loops under consideration. In this plot, a speedup of one means an execution time equal to the sequential execution time. This figure shows the result of speculatively parallelizing loops following the predictions of our framework (*Selective*) and following a policy to speculatively parallelize all the loops (*Naive*). The figure

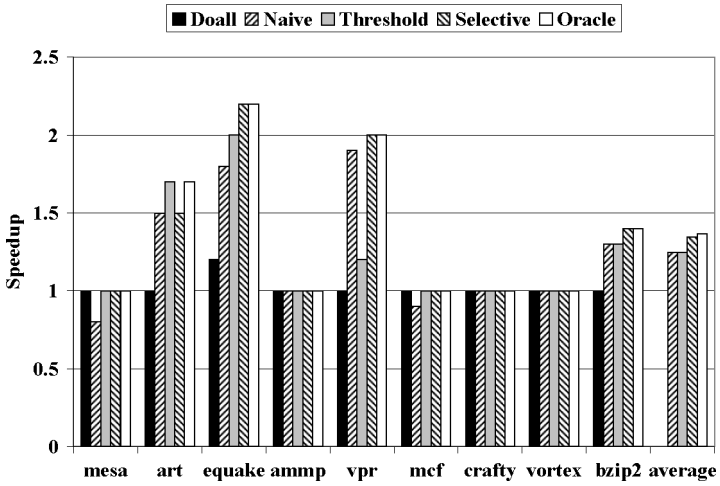

Fig. 8. Overall performance gains. Only loops that are neither Doall nor inner loops of Doall or speculatively parallelized loops are considered.

also shows the result of following a simple selection policy based on a minimum predicted thread size of 50 instructions (*Threshold*). Such policy has been used in previous work (e.g. Vijaykumar and Sohi [1998]) as a heuristic to amortize any potential overheads from speculative multithreaded execution. For reference, the figure also shows the result of parallelizing only the Doall loops (*Doall*), assuming perfect speedup for these loops (i.e., 4). Finally, the figure also shows the result of selecting the loops, based on a perfect knowledge of the expected speedups and slowdowns (*Oracle*).

From this figure, we observe that the selections based on the results of our model perform as well as or better than *Naive* for all applications, and as well as or better than *Threshold* for all applications except *179.art*. The performance gains of our model over these schemes are 8%, on average,[8] for both cases and as high as 25 and 67%, respectively. The selection policy based on our model is also able to curb the slowdowns with *Naive* in the cases of *177.mesa* and *181.mcf*. This was expected, since our model incorrectly predicts speedups in only 31% of the cases, on average (Figure 5). Finally, our selection policy achieved performance within 2% of that of *Oracle*, on average. This was expected, since our model incorrectly predicts slowdowns in less than 2% of the cases, on average (Figure 5).

5.4 Compilation Performance

In closing, in addition to evaluating the quality of our model and the impact it has on the performance of speculative parallelization, we also evaluate the performance of the model itself. Table IV shows, for each application, the wall-clock execution time of our SUIF1 pass on a 2.8-GHz Pentium 4 Xeon machine with 1 GB of main memory and running Red Hat Linux 3.2.2-5. The table also

[8]This average is computed with the simulated *sequential execution time* for each application as the weighting factor.

Table IV. Analysis Statistics and Performance[a]

Application	KLOC	Spec. pass exec. time (s)	gcc exec.time (s)	Base table size		Extd. table size	
				Avg.	Max.	Avg.	Max.
177.mesa	50	1.67	19.8	7	54	7	54
179.art	1.3	0.08	0.52	3	8	3	8
183.equake	1.5	0.14	0.65	1.8	12	1.9	12
188.ammp	13	45.57	5.38	7	97	7.8	194
175.vpr	17	3.71	5.16	3.2	9	3.2	9
181.mcf	1.9	0.04	0.95	6	16	6	16
186.crafty	21	25.56	8.93	4.8	90	6.8	162
255.vortex	53	0.62	17.86	1.5	3	1.5	3
256.bzip2	4.6	0.66	1.23	2.8	64	3.8	128

[a]Performance comparison of speculative parallelization pass and gcc and average and maximum number of different thread sizes used in the speculative parallelization cost model.

shows the number of lines of executable code, in thousands (KLOC), for each application. For comparison, the table also shows the wall-clock execution time of gcc 3.2.2 with `-DSPEC_CPU2000 -O3 -fomit-frame-pointer` flags set.

From this table we can see that, except for *188.ammp* and *186.crafty*, the execution time of the pass is comparable or shorter than a full compilation with gcc. The execution time of the speculative parallelization cost model is very reasonable, in practice, because the number of thread sizes generated in the benchmark suite is relatively small, except for the two benchmarks mentioned. The last four columns of Table IV also shows the average and maximum sizes of the base and extended thread tables. From these we can see that, on average, the number of thread sizes that the model has to manipulate (N in Section 3) is, indeed, small, although it can become large for a few loops in some applications. We also note that, with the exception of *188.ammp*, *186.crafty*, and *256.bzip2*, the addition of overhead thread sizes does not significantly increase the average and maximum number of thread sizes. This is partly because the model does not incorrectly identify too many violations.

6. RELATED WORK

Architectural support for speculative parallelization in CMP or multithreaded processors has been extensively investigated [Akkary and Driscoll 1998; Hammond et al. 1998; Krishnan and Torrellas 1999; Marcuello et al. 1998; Ooi et al. 2001; Sohi et al. 1995; Steffan and Mowry 1998; Tsai et al. 1999]. Some of these works have identified and measured the main overheads of speculative parallelization. Additional hardware-based support to reduce some of these overheads have also been investigated [Cintra and Torrellas 2002; Moshovos et al. 1997; Steffan et al. 2002], but such support tend to be costly.

Compiler technology for speculative parallelization is still a maturing field and most current techniques are based on heuristics or simple analyses. Most compilers for speculative parallelization use a simple heuristic that tries to amortize the overheads by considering only threads with estimated sizes within a certain range [Kim and Eigenmann 2001; Vijaykumar 1998; Vijaykumar and Sohi 1998]. To identify sources of speculative buffer overflow, the work

in Kim and Eigenmann [2001] uses cache-miss equations to statically detect potential conflicts. Cache-miss equations work well for affine array references, but are not well suited for more irregular access patterns. The work in Zhai et al. [2002, 2004] explores optimized compiler scheduling of instructions to reduce the critical communication path between speculative threads, for both register- and memory-resident values. Probabilistic dependence analysis has been used in Chen et al. [2003], and Zhai et al. [2004] to estimate the likelihood of data-dependence violations, while a simple count of possible cross-thread dependences has been used in Bhowmik and Franklin [2002]. To the best of our knowledge, no previous work has investigated a compiler framework that models all speculative parallelization overheads and attempts to quantitatively estimate the performance gain, or loss, of the speculative execution. Concurrently with our work, Du et al. [2004] developed a quantitative cost model to estimate the performance loss of speculative parallelization in the presence of data-dependence violations and squashes. Our works differ in that our proposed framework attempts to accommodate all speculative parallelization overheads and, in this paper, we focus on the load imbalance overhead.

In addition to static compiler optimizations, others have investigated the use of profiling to identify good thread partitioning for speculative execution [Liu et al. 2005; Marcuello and González 2002; Olukotun et al. 1999; Whaley and Kozyrakis 2005]. Alternatively, dynamic partitioning of threads with on-the-fly performance information through hardware monitors has been proposed in Chen and Olukotun [2003], and Warg and Stenstrom [2003]. Both approaches usually consider all the overheads combined, but require either feedback-directed or dynamic recompilation of the code. There has also been some work on compiler support for speculative multithreading based on "helper threads" [Quinones et al. 2005]. This model of speculative multithreading differs significantly from speculative parallelization and leads to very different cost models and optimizations.

Perhaps the closest work to ours is Quinones et al. [2005], which also analyzes all control flow paths and attempts to estimate the performance gains from speculative multithreading. With respect to the execution and cost model, that work differs from ours in that it attempts to emulate a trace-driven-like execution of the threads using the control, flow graph, while, in our work, we attempt to model the execution of threads mathematically. Another important difference is with respect to the way threads are generated from the sequential code. While we only consider loop iterations as threads, that work considers threads created from basic blocks between what are called "control quasiindependent points." Finally, unlike ours, that work also considered the possibility, and costs, of adding precomputation slices to minimize some of the squash and communication overheads.

7. CONCLUSIONS

In this paper we have proposed and evaluated a model of speculative multithreaded execution that can be used by the compiler to reason about the

overheads and expected resulting performance gains, or losses, from speculative parallelization. This model is based on estimating the likely execution duration of threads and properly takes into account the scheduling restrictions of most speculative execution environments. In this way, the model is flexible enough to include all speculative parallelization overheads. Different from previous work, which present heuristics that attempt to estimate "good" or "bad" sections for speculative multithreaded execution, our compiler framework attempts to quantitatively estimate the speedup or slowdown. Such an estimate can then be used by the compiler or runtime system to make more complex and educated tradeoff decisions. For instance, in a highly loaded multiprogrammed environment, the compiler or runtime system may decide to switch off speculative parallelization even when a speedup is expected, if this speedup is too small and does not justify the use of the extra resources.

The model proposed requires data structures that are simple to generate and manage and formulas that are fast to compute. Where compile-time information is too incomplete, the accuracy of the model could also be improved with only simple profile information that can be obtained from a sequential execution of the program. Finally, the model can be easily added to existing compiler frameworks and requires little, if any, modification to common intermediate representations.

Experimental results on a number of loops from SPEC benchmarks show that our framework can identify, on average, 53% of the loops that cause slowdowns and, on average, 95% of the loops that lead to speedups. In fact, our framework predicts the speedups with an error of less than 20% for an average of 33% of the loops across the benchmarks and with an error of less than 50% for an average of 79% of the loops. Overall, the framework often outperforms, by as much as 38%, a naive approach that attempts to speculatively parallelize all the loops considered and is able to curb the large slowdowns caused, in many cases, by this naive approach.

APPENDIX

An Example Computation of S_{est}

As an example, consider the code section in Figure 3a. The corresponding base table is shown in Table AI. Thus, $B = \{W_1, W_2\}$. Assume a system with four processors.

Looking at the memory operations associated with W_2, we can assume the possibility of a data-dependence violation when two different *instances* of this thread size appear in the same tuple and the address of the store of the predecessor is the same as the address of the load of the successor. This leads to a new possible thread size W_3, as shown in Table AII and Figure 3c. For the sake of the example, assume that the probability that the same memory addresses are accessed by two instances of W_2 is $p_{dep} = 0.1$.

Looking at the memory operations associated with W_1, we can also assume the possibility of a speculative buffer overflow at the store of this thread. This leads to two new possible thread sizes W_4, when the predecessor to wait for is an

Table AI. Base Thread Sizes, Probabilities, and Memory Operations with Timestamps (i.e., base table) for the Example in Figure 3a[a]

Thread name	Probability of occurrence (p_i)	Execution time	Memory operations	Example parameters
W_1	50%	$w1+w5+w6$	st_Y@T_1	$W=100; T_1=50$
W_2	50%	$w1+w2+w3+w4$	ld_X@T_2 st_X@T_3	$W=200$ $T_2=30; T_3=120$

[a]The thread sizes are sorted in increasing order (thus $w2+w3+w4 > w5+w6$). The values in the last column are arbitrary and only for the sake of the example

Table AII. Base and Extended Thread Sizes (i.e., Extended Table) for the Example in Figure 3a

Thread name	Final rank	Execution time	Example size
W_1	1	$w1+w5+w6$	100
W_2	3	$w1+w2+w3+w4$	200
W_3	5	$2*(w1+w2+w3)+w4$	320
W_4	2	$(w1+w5+w6)+w6$	150
W_5	4	$(w1+w2+w3+w4)+w6$	250

instantiation of W_1, and W_5, when the predecessor to wait for is an instantiation of W_2, as shown in Table AII and Figure 3d. For the sake of the example, assume that the probability that the store will overflow the speculative buffer is $p_{ovflow} = 0.3$. Thus, $E = \{W_1, W_4, W_2, W_5, W_3\}$.

The next step is to compute the terms $p_{i,j}$, the probabilities that thread size W_i appears in processor j in a given tuple. The terms for $i = 2$ and $i = 3$ decrease in Case 2 of Section 3.3, as W_2 is associated with a squash overhead. Thus, using Eq. (8) and substituting both p_{prod} and p_{base} with p_2:

$$
\begin{aligned}
p_{2,0} &= (1-p_2)^0 * p_2 + (1-(1-p_2)^0) * p_2 * (1-p_{dep}) = 0.5 \\
p_{2,1} &= (1-p_2)^1 * p_2 + (1-(1-p_2)^1) * p_2 * (1-p_{dep}) = 0.475 \\
p_{2,2} &= (1-p_2)^2 * p_2 + (1-(1-p_2)^2) * p_2 * (1-p_{dep}) = 0.4625 \\
p_{2,3} &= (1-p_2)^3 * p_2 + (1-(1-p_2)^3) * p_2 * (1-p_{dep}) = 0.45625
\end{aligned}
$$

$$
\begin{aligned}
p_{3,0} &= (1-(1-p_2)^0) * p_2 * p_{dep} = 0 \\
p_{3,1} &= (1-(1-p_2)^1) * p_2 * p_{dep} = 0.025 \\
p_{3,2} &= (1-(1-p_2)^2) * p_2 * p_{dep} = 0.0375 \\
p_{3,3} &= (1-(1-p_2)^3) * p_2 * p_{dep} = 0.04375
\end{aligned}
$$

Note that for every j, the sum of $p_{2,j}$ and $p_{3,j}$ equals $p_2 = 0.5$, as expected. Also note that for increasing j the value of $p_{2,j}$ decreases and the value of $p_{3,j}$ increases. This is also expected as the chances of a data-dependence violation increases with more predecessors. This behavior is not captured in the model of Dou and Cintra [2004] and in any of the current compiler models.

The terms for $i = 1$, $i = 4$, and $i = 5$ decrease in Case 3 of Section 3.3, as W_1 is associated with a speculative buffer overflow overhead. Thus, using Eq. (9)

and substituting p_{base} with p_1, $p_{firstlonger}$ with p_1, $p_{waitfor}$ with p_1 for W_4, and $p_{waitfor}$ with p_2 for W_5:

$$p_{1,0} = p_1 * (1 - p_{ovflow}) + p_1 * p_{ovflow} * \left(1 - \sum_{k=1}^{2} p_k\right)^0 = 0.5$$

$$p_{1,1} = p_1 * (1 - p_{ovflow}) + p_1 * p_{ovflow} * \left(1 - \sum_{k=1}^{2} p_k\right)^1 = 0.35$$

$$p_{1,2} = p_1 * (1 - p_{ovflow}) + p_1 * p_{ovflow} * \left(1 - \sum_{k=1}^{2} p_k\right)^2 = 0.35$$

$$p_{1,3} = p_1 * (1 - p_{ovflow}) + p_1 * p_{ovflow} * \left(1 - \sum_{k=1}^{2} p_k\right)^3 = 0.35$$

$$p_{4,0} = \left(\left(\sum_{k=1}^{1} p_k\right)^0 - \left(\sum_{k=1}^{0} p_k\right)^0\right) * p_{ovflow} * p_2 = 0$$

$$p_{4,1} = \left(\left(\sum_{k=1}^{1} p_k\right)^1 - \left(\sum_{k=1}^{0} p_k\right)^1\right) * p_{ovflow} * p_2 = 0.075$$

$$p_{4,2} = \left(\left(\sum_{k=1}^{1} p_k\right)^2 - \left(\sum_{k=1}^{0} p_k\right)^2\right) * p_{ovflow} * p_2 = 0.0375$$

$$p_{4,3} = \left(\left(\sum_{k=1}^{1} p_k\right)^3 - \left(\sum_{k=1}^{0} p_k\right)^3\right) * p_{ovflow} * p_2 = 0.01875$$

$$p_{5,0} = \left(\left(\sum_{k=1}^{2} p_k\right)^0 - \left(\sum_{k=1}^{1} p_k\right)^0\right) * p_{ovflow} * p_2 = 0$$

$$p_{5,1} = \left(\left(\sum_{k=1}^{2} p_k\right)^1 - \left(\sum_{k=1}^{1} p_k\right)^1\right) * p_{ovflow} * p_2 = 0.075$$

$$p_{5,2} = \left(\left(\sum_{k=1}^{2} p_k\right)^2 - \left(\sum_{k=1}^{1} p_k\right)^2\right) * p_{ovflow} * p_2 = 0.1125$$

$$p_{5,3} = \left(\left(\sum_{k=1}^{2} p_k\right)^3 - \left(\sum_{k=1}^{1} p_k\right)^3\right) * p_{ovflow} * p_2 = 0.13125$$

Note that for every j, the sum of $p_{1,j}$, $p_{4,j}$, and $p_{5,j}$ equals $p_1 = 0.5$, as expected. Also note that for every j $p_{4,j} \leq p_{5,j}$. This is also expected, as for W_4 to appear, it is necessary that W_2 does not appear in any predecessor processor, while W_5 can appear even if W_1 appears in several predecessor processors, as long as W_2 appears in at least one predecessor processor.

Now using Eq. (11) we compute the probabilities of the parallel execution time of a given thread tuple i (the correspondence between $W_{rank(j)}$ and some W_k are given in Table AII):

$$p(Tpar_{tuple_i} = W_{rank(1)}) = \prod_{k=0}^{3}\left(\sum_{l=1}^{1} p_{rank(l),k}\right) - \sum_{m=1}^{0} p(Tpar_{tuple_{i'}} = W_{rank(m)}) = 0.0214$$

$$p(Tpar_{tuple_i} = W_{rank(2)}) = \prod_{k=0}^{3}\left(\sum_{l=1}^{2} p_{rank(l),k}\right) - \sum_{m=1}^{1} p(Tpar_{tuple_{i'}} = W_{rank(m)}) = 0.009$$

$$p(Tpar_{tuple_i} = W_{rank(3)}) = \prod_{k=0}^{3}\left(\sum_{l=1}^{3} p_{rank(l),k}\right) - \sum_{m=1}^{2} p(Tpar_{tuple_{i'}} = W_{rank(m)}) = 0.6007$$

$$p(Tpar_{tuple_i} = W_{rank(4)}) = \prod_{k=0}^{3}\left(\sum_{l=1}^{4} p_{rank(l),k}\right) - \sum_{m=1}^{3} p(Tpar_{tuple_{i'}} = W_{rank(m)}) = 0.2663$$

$$p(Tpar_{tuple_i} = W_{rank(5)}) = \prod_{k=0}^{3}\left(\sum_{l=1}^{5} p_{rank(l),k}\right) - \sum_{m=1}^{4} p(Tpar_{tuple_{i'}} = W_{rank(m)}) = 0.1026$$

Note that for every j the sum of $p(Tpar_{tuple_i} = W_j)$ equals 1, as expected. Also note that $p(Tpar_{tuple_i} = W_2)$ and $p(Tpar_{tuple_i} = W_5)$ are by far the largest terms. The first is expected, since W_2 is a base size that is only replaced with a small probability (p_{dep}). The later occurs because, despite W_5 having smaller probabilities of appearing in any given processor than its corresponding base thread size, i.e. $p_{5,j} < p_{1,j}$ for every j, W_5 has a larger probability of deciding the parallel time since it is much larger than W_1. Finally, using Eq. (5), (6), and

(12), we compute $Tseq_{est}$, $Tpar_{est}$, and S_{est}:

$$Tseq_{est} = 4\sum_{i=1}^{2} W_i p_i = 600$$

$$Tpar_{est} = \sum_{j=1}^{5} W_j p(Tpar_{tuple_i} = W_j) = 223$$

$$S_{est} = \frac{Tseq_{est}}{Tpar_{est}} = \frac{600}{223} = 2.69$$

REFERENCES

AKKARY, H. AND DRISCOLL, M. A. 1998. A dynamic multithreading processor. In *International Symposium on Microarchitecture*. 226–236.

BHOWMIK, A. AND FRANKLIN, M. 2002. A general compiler framework for speculative multithreading. In *Symposium on Parallel Algorithms and Architectures*. 99–108.

CHATTERJEE, S., PARKER, E., HANLON, P. J., AND LEBECK, A. R. 2001. Exact analysis of the cache behavior of nested loops. In *Conference on Programming Language Design and Implementation*. 286–297.

CHEN, M. AND OLUKOTUN, K. 2003. Test: a tracer for extracting speculative threads. In *International Symposium on Code Generation and Optimization*. 301–312.

CHEN, P.-S., HUNG, M.-Y., HWANG, Y.-S., JU, R. D.-C., AND LEE, J. K. 2003. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In *Symposium on Principles and Practice of Parallel Programming*. 24–36.

CINTRA, M. AND TORRELLAS, J. 2002. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *International Symposium on HighPerformance Computer Architecture*. 43–54.

COLOHAN, C. B., AILAMAKI, A., STEFFAN, J. G., AND MOWRY, T. C. 2006. Tolerating dependences between large speculative threads via sub-threads. In *International Symposium on Computer Architecture*. 216–226.

DOU, J. AND CINTRA, M. 2004. Compiler estimation of load imbalance overhead in speculative parallelization. In *International Conference on Parallel Architectures and Compilation Techniques*. 203–214.

DU, Z.-H., LIM, C.-C., LI, X.-F., YANG, C., ZHAO, Q., AND NGAI, T.-F. 2004. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Conference on Programming Language Design and Implementation*. 71–81.

DUBEY, P., O'BRIEN, K., O'BRIEN, K. M., AND BARTON, C. 1995. Single-program speculative multithreading (spsm) architecture: Compiler-assisted fine-grained multithreading. In *International Conference on Parallel Architectures and Compilation Techniques*. 109–121.

GARZARÁN, M. J., PRVULOVIC, M., LLABERÍA, J. M., VINALS, V., RAUCHWERGER, L., AND TORRELLAS, J. 2003. Tradeoffs in buffering memory state for thread-level speculation in multiprocessors. In *International Symposium on High-Performance Computer Architecture*. 191–202.

GIRKAR, M. AND POLYCHRONOPOULOS, C. D. 1994. The hierarchical task graph as a universal intermediate representation. *International Journal of Parallel Programming 22,* 5 (Oct.).

HALL, M., ANDERSON, J., AMARASINGHE, S., MURPHY, B., LIAO, S.-W., BUGNION, E., AND LAM, M. 1996. Maximizing multiprocessor performance with the suif compiler. *IEEE Computer 29,* 12 (Dec.), 84–89.

HAMMOND, L., WILEY, M., AND OLUKOTUN, K. 1998. Data speculation support for a chip multiprocessor. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 58–69.

HAMMOND, L., HUBBERT, B., SIU, M., PRABHU, M., CHEN, M., AND OLUKOTUN, K. 2000. The stanford hydra cmp. *IEEE Micro 12,* 2 (Mar.), 71–84.

KIM, S. W. AND EIGENMANN, R. 2001. The structure of a compiler for explicit and implicit parallelism. In *International Workshop on Languages and Compilers for Parallel Computing*.

KleinOsowski, A. J. and Lilja, D. J. 2002. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters 1*.

Krishnan, V. and Torrellas, J. 1999. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers, Special Issue on Multithreaded Architectures 48,* 9 (Sept.), 866–880.

Li, Z., Tsai, J.-Y., Wang, X., Yew, P.-C., and Zheng, B. 1996. Compiler techniques for concurrent multithreading with hardware speculation support. In *International Workshop on Languages and Compilers for Parallel Computing*.

Liu, W., Tuck, J., Ceze, L., Strauss, K., Renau, J., and Torrellas, J. 2005. Posh: A profiler-enhanced tls compiler that leverages program structure. In *Watson Conference on Interaction between Architecture, Circuits, and Compilers*.

Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., and Werner, B. 2002. Simics: A full system simulation platform. *IEEE Computer 35,* 2 (Feb.), 50–58.

Marcuello, P., González, A., and Tubella, J. 1998. Speculative multithreaded processors. In *International Conference on Supercomputing*. 77–84.

Marcuello, P. and González, A. 2002. Thread-spawning schemes for speculative multithreading. In *International Symposium on High-Performance Computer Architecture*. 55–64.

Moshovos, A., Breach, S. E., Vijaykumar, T. N., and Sohi, G. S. 1997. Dynamic speculation and synchronization of data dependences. In *International Symposium on Computer Architecture*. 181–193.

Olukotun, K., Hammond, L., and Willey, M. 1999. Improving the performance of speculatively parallel applications on the hydra cmp. In *International Conference on Supercomputing*. 21–30.

Ooi, C.-L., Kim, S. W., Park, I., Eigenmann, R., Falsafi, B., and Vijaykumar, T. N. 2001. Multiplex: Unifying conventional and speculative thread-level parallelism on a chip multiprocessor. In *International Conference on Supercomputing*. 368–380.

Oplinger, J., Heine, D., and Lam, M. 1999. In search of speculative thread-level parallelism. In *International Conference on Parallel Architectures and Compilation Techniques*. 303–313.

Patterson, J. R. C. 1995. Accurate static branch prediction by value range propagation. In *Conference on Programming Language Design and Implementation*. 67–78.

Prabhu, M. K. and Olukotun, K. 2003. Using thread-level speculation to simplify manual parallelization. In *Symposium on Principles and Practice of Parallel Programming*. 1–12.

Quinones, C. G., Madriles, C., Sánchez, J., Marcuello, P., González, A., and Tullsen, D. M. 2005. Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. In *Conference on Programming Language Design and Implementation*. 269–279.

Renau, J., Tuck, J., Liu, W., Ceze, L., Strauss, K., and Torrellas, J. 2005. Tasking with out-of-order spawn in tls chip multiprocessors: Microarchitecture and compilation. In *International Conference on Supercomputing*. 179–188.

Sohi, G., Breach, S., and Vijaykumar, T. N. 1995. Multiscalar processors. In *International Symposium on Computer Architecture*. 414–425.

Steffan, J. G. and Mowry, T. C. 1998. The potential for using thread-level data speculation to facilitate automatic parallelization. In *International Symposium on High-Performance Computer Architecture*. 2–13.

Steffan, J. G., Colohan, C. B., Zhai, A., and Mowry, T. C. 2002. Improving value communication for thread-level speculation. In *International Symposium on High-Performance Computer Architecture*. 65–75.

Tsai, J.-Y., Huang, J., Amlo, C., Lilja, D., and Yew, P.-C. 1999. The superthreaded processor architecture. *IEEE Transactions on Computers, Special Issue on Multithreaded Architectures 48,* 9 (Sept.), 881–902.

Vera, X. and Xue, J. 2002. Let's study whole-program cache behavior analitically. In *International Symposium on High-Performance Computer Architecture*. 176–185.

Vijaykumar, T. N. 1998. Compiling for the multiscalar architecture. Ph.D. thesis, University of Wisconsin at Madison.

Vijaykumar, T. N. and Sohi, G. S. 1998. Task selection for a multiscalar processor. In *International Symposium on Microarchitecture*. 81–92.

WAGNER, T. A., MAVERICK, V., GRAHAM, S. L., AND HARRISON, M. A. 1994. Accurate static estimators for program optimization. In *Conference on Programming Language Design and Implementation*. 85–96.

WARG, F. AND STENSTROM, P. 2003. Improving speculative thread-level parallelism through module run-length prediction. In *International Parallel and Distributed Processing Symposium*. 12–19.

WHALEY, J. AND KOZYRAKIS, C. 2005. Heuristics for profile-driven method-level speculative parallelization. In *International Conference on Parallel Computing*.

ZHAI, A., COLOHAN, C. B., STEFFAN, J. G., AND MOWRY, T. C. 2002. Compiler optimization of scalar value communication between speculative threads. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 171–183.

ZHAI, A., COLOHAN, C. B., STEFFAN, J. G., AND MOWRY, T. C. 2004. Compiler optimization of memory-resident value communication between speculative threads. In *International Symposium on Code Generation and Optimization*. 39–50.

Received March 2006; revised July 2006; accepted October 2006