Reducing Misspeculation Overhead for Module-Level Speculative Execution

Fredrik Warg
warg@ce.chalmers.se

Per Stenstrom
pers@ce.chalmers.se

Department of Computer Science and Engineering
Chalmers University of Technology
Göteborg, Sweden

ABSTRACT

Thread-level speculative execution is a technique that makes it possible for a wider range of single-threaded applications to make use of the processing resources in a chip multiprocessor.

We consider module-level speculation, i.e., speculative threads executing the code after a module (i.e., a procedure, function, or method) call. Unfortunately, previous studies have shown that indiscriminate module-level speculation results in significant overheads, mainly due to frequent misspeculations. In addition to hurting performance, excessive overhead is harmful from a resource usage and energy efficiency standpoint. We show that the overhead when spawning speculative threads for all module continuations is on average three times as big as the time spent on useful execution on our baseline 8-way chip multiprocessor.

In this paper, we present and make a detailed evaluation of a technique that aims at reducing the overhead associated with misspeculations. History-based prediction is used in an attempt to prevent speculative threads from being spawned when they are expected to cause misspeculations. We find that the overhead can be reduced with a factor of six on average compared to indiscriminate speculation. The impact on speedup is small for most applications, but in several cases speedup is slightly improved.

Categories and Subject Descriptors

C.1.2 [**Computer Systems Organization**]: Processor Architectures—*Multiple Data Stream Architectures (Multiprocessors)*; C.4 [**Computer Systems Organization**]: Performance of Systems

General Terms

Design, Measurement, Performance

Keywords

Thread-Level Speculation, Module-Level Parallelism, Chip Multiprocessors, Misspeculation Prediction, Performance Evaluation

1. INTRODUCTION

Chip multiprocessors (CMPs) are an emerging architectural style owing to diminishing returns in exploiting instruction-level parallelism, and the increased complexity of the superscalar paradigm. CMPs exploit thread-level parallelism in addition to the instruction-level parallelism targeted by superscalar processor cores. Unfortunately, most applications are single-threaded and thus a bad match for CMPs.

Speculative thread-level parallelism (STLP) provides support for aggressive automatic thread-level parallelization by relaxing the demand that threads are provably data independent. Instead, a speculation system makes sure that data dependence violations are detected and misspeculations resolved. Since chip multiprocessors are tightly integrated, they are a good match for executing fine-grained speculative threads. Consequently, there is a multitude of proposals for integrating a speculation system with a chip multiprocessor [5, 6, 7, 9, 12, 15, 17, 18].

There are several approaches to create threads for an STLP machine, for instance using loop iterations or module (i.e. procedure, function, or method) continuations – the code after the call is run in parallel with the called module. Since one of our main goals is to explore techniques that can be used for run-time parallelization, module-level parallelism (MLP) is an appealing option; modules are particularly easy to identify at run-time. Module-level parallelism has been investigated in [2, 6, 14, 19].

The drawback with aggressive module-level speculation, i.e. creating speculative threads for all module continuations, is that the overhead can easily dominate the execution time, preventing us from achieving the best possible speedups, and in some cases even cause slowdown compared to sequential execution.

Overhead is defined as all extra work associated with events that do not occur in the sequential execution of the program. One type of overhead is all work the speculation system performs in order to manage speculative threads. This thread-management overhead consists of thread-start, roll-back, and commit (thread completion) overhead. Some overhead is compulsory and equal for all threads. Its impact is therefore proportional to the size of the overhead compared to the size of the speculative threads – thread-start and commit belongs to this category.

Another type of overhead is related to the actual program being executed. When threads are rolled back due to a misspeculation, the work done by the squashed threads is thrown away, this is execution overhead. In addition, some memory accesses and interthread communication proved to be redundant, the communication overhead. Overhead related to misspeculations does, in contrast to the compulsory overhead, not contribute to our sought-after parallelism. This overhead is harmful for several reasons. First, threads that are squashed will have occupied processing and communication resources, as well as storage space for its speculative state, without contributing to the forward progress of the program. In fact, this can potentially hamper successful threads, thereby slowing down execution. Second, even if there are plenty of free resources, overhead is a serious drawback when considering energy-efficiency or the resource usage in a multiprogrammed environment.

In this paper we propose a *misspeculation prediction* technique which tries to avoid creating threads that will misspeculate. The method uses history-based prediction, i.e. based on previous violations, when deciding whether to create a new thread or not, and can be integrated with a speculation run-time system. Our goal is two-fold: we want to reduce the total overhead, and if possible improve speedup, compared to naively spawning new speculative threads for all module calls.

We investigate a number of predictors and different ways to record misspeculations, and find that using a simple last-outcome predictor indexed with module ID can bring down the overhead a factor of six compared to indiscriminate speculation. Speedup is improved for four applications, but noticeable worse for two applications.

There have been other attempts to avoid misspeculations. In [4, 11, 16] the approach is to record cross-thread dependences and synchronize dependent load-store pairs. However, the threads will still incur thread-start and commit overheads. In contrast, threads that are expected to misspeculate will not be created with our approach, which means thread-start and commit overhead is avoided.

The applications that do not benefit from misspeculation prediction are those who suffer from few misspeculations to begin with. By applying misspeculation prediction selectively, i.e. only when the ratio of squashes compared to new thread starts is above a certain threshold (0.6 works well for our applications), we avoid the negative impact on speedup at the expense of slightly higher overhead. This method gives the same or slightly higher speedup for all applications compared to indiscriminate module-level speculation, but with almost four times lower average overhead.

We begin by explaining the execution model and presenting the simulation tools in Section 2. Section 3 introduces the misspeculation prediction technique, and in Section 4 we investigate a number of implementation design choices. Then, misspeculation prediction is improved upon with a method for selective use in Section 5. Related works are discussed in Section 6, and finally, we conclude in Section 7.

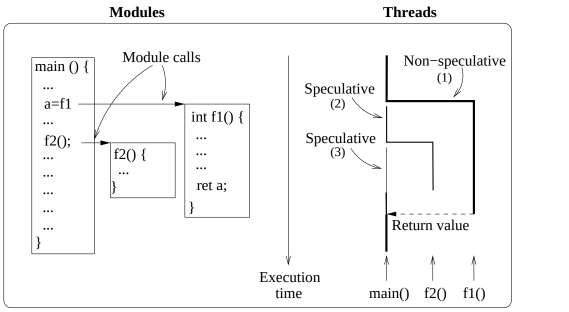
2. METHODOLOGY

In this section we present the baseline system and its module-level speculative execution model, our experimental methodology, as well as the benchmark applications used throughout the paper.

2.1 Baseline Execution Model

In the thread-level speculation model, multiple threads derived from the same sequential code are executed in parallel on their own processor cores. As opposed to other parallelization techniques, there are no guarantees that the threads are independent. Instead, a *speculation system* enforces data dependences. A dependence violation causes a misspeculation, and in order to recover one or more dependent threads need to be squashed and re-executed.

The threads are ordered according to their relative position in a sequential execution of the program; a thread is said to be more speculative than another thread if it would execute after it in the sequential case. For instance, in Figure 1, thread (3) is more speculative than thread (2). Thread (1) is called the non-speculative thread since there are no less speculative threads in the system; the non-speculative thread never has to be squashed since there remains no unresolved dependences with earlier threads, thus forward progress is guaranteed as long as the non-speculative thread is allowed to execute.



```
main () {
  ...
  a=f1
  ...
  f2();
  ...
  ...
  ...
  ...
  ...
}
```

```
f2() {
  ...
}
```

```
int f1() {
  ...
  ...
  ...
  ret a;
}
```

Figure 1: New speculative threads are spawned for module continuations. An example program is shown to the left and the corresponding thread tree to the right.

A misspeculation occurs if there is a flow dependency between two threads and the more speculative thread reads the shared variable before the less speculative thread has written it. A flow dependence where the less speculative thread produces the data before the more speculative thread needs it can be resolved by forwarding speculative data from less to more speculative threads. Other types of dependences (anti and output) are resolved through renaming by the speculation system.

The speculation system takes care of spawning threads, checking for data dependences, forwarding data between threads, and rolling back execution to a correct state if a misspeculation occurs. As long as a thread is speculative, the results produced can not be stored in a non-reversible way, since it is possible that execution must roll back. Most proposed STLP machines store results in special-purpose buffers or in a modified cache hierarchy until the thread is non-speculative. Results produced by speculative threads and accompanying state information is called the speculative state. When the non-speculative thread has completed execution, it can commit; that is, results are merged with main memory state and the thread is retired from the speculation system. The buffers must also keep track of the ordering among versions of the same address, in order to facilitate dependence detection. If a violation occurs, the offending thread and its successors are squashed, i.e. all their speculative results are thrown away and the threads restarted. Exceptions and I/O operations must be executed non-speculatively since speculative threads are not allowed to alter the system state permanently in any way.

In addition, register-level dependences between threads need to be resolved. One option is to avoid allocating global variables in registers at compile-time. Other options that do not require recompilation is to use binary translation and additional hardware to handle cross-thread register dependences [7] or, for module-level parallelism, a mechanism to detect register dependences before committing a thread can be used [13]. In our simulations, global variables are forced to always access memory; thus, all dependences are handled by the versioning system for memory accesses.

Module-level speculation treats module calls as potential points where a new speculative thread can be spawned. When encountering a call instruction, the original thread will continue to execute the called function and, if the called module is believed to contribute to speedup if run in parallel with the subsequent code, a new thread is created for execution of the module continuation (i.e. the code after the call instruction). Thread creation is shown in Figure 1. The new thread will get its initial state, including register contents and starting address, from the original thread. A new thread will be more speculative than its parent, but inherits the relationship of its parent with respect to all other threads. That means in module-level speculation the most recently created thread is not necessarily the most speculative one.

2.2 Speculation Overhead

Since the mechanism presented in this paper aims at reducing overhead, a closer look at the different sources of overhead is needed. As mentioned, all extra work that do not occur in a sequential execution of the program is defined as overhead and is harmful for several reasons.

The *thread-management overhead* is compulsory and consists of thread-start and commit, the time it takes to start and finalize a thread respectively. The compulsory overhead is needed to extract thread-level parallelism. If, for instance, thread-start is time-consuming, the benefit from starting a new thread in terms of parallel execution of useful code will suffer. The impact can be kept under control with efficient speculation mechanisms and by avoiding to create too small threads. In [20] we presented a method that will prevent small modules from being used for speculation.

We also include roll-back overhead in the general category of thread-management overheads, since it is controlled by the speculation system. However, it is not compulsory. Roll-backs occur as a result of a misspeculation, and thus can be avoided if we find a method to avoid the dependence violation that caused the misspeculation.

In addition to the roll-back handler overhead, partially completed threads that were squashed due to a misspeculation need to be re-executed. The *execution overhead* is work done by threads that had to be squashed. A related type is *communication overhead*, or the extra memory accesses and intra-processor communication caused by threads that are later squashed. Communication overhead could be seen as part of the execution overhead, but deserves special mention since it potentially hampers other threads competing for the same resources.

Figure 2 shows two example execution snapshots. To the left, a successful speculative thread with only compulsory overhead in the form of a thread-start (commit is omitted in the figure to reduce the clutter, but is located at the end of every successfully finished thread). In the example to the right, however, the speculative thread suffered from a dependence violation and had to be restarted. In addition to thread-start, we also have execution-, rollback-, and communication overhead.

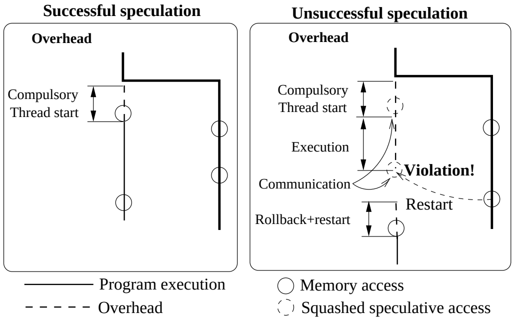


Figure 2: Sources of overhead in thread-level speculation.

In summary: *Total overhead = thread-start + commit + rollback + execution + communication*. Throughout the rest of the paper, we refer to the total overhead unless otherwise stated. The simulation results measure the total overhead as the percentage of extra cycles incurred by all forms of overhead added together compared to the number of cycles used in a sequential execution of the application.

2.3 Simulation Model

Experimental Approach

A high-level view of the experimental approach is shown in Figure 3. We first annotate the program at each module call and return, and whenever a return value is used. We have done this by slightly modifying GCC 2.95.2. Since only modules in the application are annotated, library calls are never run speculatively in our simulations. The binaries are run on a Linux system on top of a SPARC-based instruction-set simulator using Simics [8]. The annotations, as well as reads and writes, trigger a module attached to Simics which in turn generates a trace of events. Each event is associated with a time-stamp from a timer measuring elapsed clock-cycles in the simulated machine. This gives us an accurate sequential execution trace of the application.

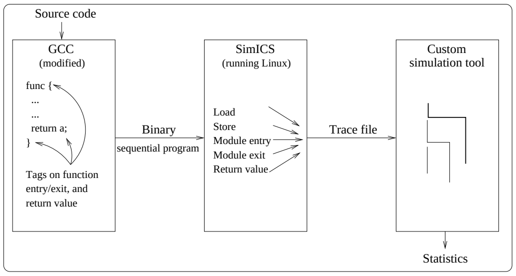


Figure 3: The GCC, Simics, and STLP simulator tool-chain.

The sequential trace is fed into our speculative CMP simulation tool (called Custom simulation tool in Figure 3). This tool simulates program execution according to the baseline system with indiscriminate speculation as well as systems with the misspeculation prediction techniques investigated later in this paper. Additional parameters of the baseline system are summarized in Table 1.

The target machine is modeled without actually executing any code. Instead, the time-stamps from the trace are used to model the execution time elapsed in between simulated events. Therefore the processor model is determined by Simics and not our tool. Due to a relatively simple architectural model with ideal communication, the time-stamps from the sequential execution will reflect elapsed time in the parallel case as well.

Justification of simulation model

The key metric of comparing the baseline system with the enhanced systems is execution overhead. As a result, we opted for using a simulation model that makes a few simplifying assumptions that we do not expect to qualitatively affect the conclusions. The first simplifying assumption is the use of an ideal memory system in which all memory accesses take a single cycle to be performed. The second simplifying assumption is the use of single-issue in-order processors. Let's discuss these simplifying assumptions in turn.

A detailed memory system model, in contrast to ours, would capture the communication overhead mentioned in Section 2.2. The performance of the baseline system as well as the system with

Table 1: Baseline speculative chip multiprocessor.

Processor cores	8-way chip multiprocessor with single-issue in-order cores and 1-cycle memory accesses (ie CPI=1). The 8-way configuration is based on the findings in [19]; the technique is also verified with a 4-way machine.
Overhead	Thread-starts, roll-backs, and context switches are modeled with fixed-length 100-cycle overhead. The technique is also verified with 50- and 10-cycle overheads.
Value prediction	Stride value prediction is used for module return values; this was found to be effective in [19].
Thread scheduling	Threads can be preempted. On an *N*-way machine the *N* least speculative threads are always scheduled to run. There is no limit on the number of threads active and waiting to run.
Data dependences	Anti- and output dependences solved with renaming; flow dependences solved with forwarding when possible. Dependences which can not be solved causes a roll-back and restart of the dependent thread and all its child threads (ie threads that are spawned from the dependent thread). Buffer space for speculative state is unlimited.

Table 2: Benchmark applications.

Name	*Origin*	*Description*	*#Instructions (dynamic)*	*#Modules (dynamic)*	*Avg. instr./mod. (dynamic)*	*#Modules (static)*
gcc	SPECint95	GNU C Comp. 2.5.3	13M	54.5k	237	525
compress	SPECint95	Unix compress	1.4M	21k	67	8
db	SPEC JVM98	Simple database	13M	4.9k	2644	52
deltablue	Sun Labs	Constraint solver	2.6M	12.5k	208	76
go	SPECint95	The game of Go	1.4M	1.1k	1190	105
idea	jBYTEmark	En/decryption	35.7M	12k	2966	16
jess	SPEC JVM98	Expert system	16.3M	25.8k	633	484
m88ksim	SPECint95	A chip simulator	2.2M	0.5k	4767	34
neuralnet	jBYTEmark	Neural network	4.2M	2.6k	1626	26

misspeculation prediction support we propose in the next section would be negatively affected by additional communication overhead. However, since the baseline system will incur more misspeculations, the overhead associated with the memory system will be higher. Thus, assuming an ideal memory system will in that respect give an advantage to the baseline system. One could argue that misspeculations could result in useful prefetching of data to subsequently executed threads [3]. With our simulation setup we are of course unable to quantify the impact on performance of this effect.

As for assuming single-issue processors, it is an interesting design point for chip multiprocessors; In the Piranha [1] architecture, single issue processors have been proposed as they are a better match to server type applications. A superscalar core would introduce additional aspects, such as a tradeoff between instruction- and thread-level parallelism. We note that it is outside the scope of the paper to study this tradeoff.

2.4 Benchmarks

Table 2 summarizes the benchmark applications. These applications were used in our previous studies [19, 20], and some of them have also been used in earlier STLP limit studies [2, 10, 14]. Thus they are chosen mainly for reasons of comparison. The input data sets are quite small; they have been reduced due to limitations in our simulator, but the number of dynamic as well as static modules in each trace is still considerable as shown in Table 2.

We have concentrated on integer programs known to be hard to parallelize with parallelizing compilers, and which do not contain an abundance of loop-level parallelism such as, for instance, in SPECfp. Parallelism in these applications is harder to exploit, and will likely require a tightly integrated STLP machine with low communication latencies – a chip multiprocessor or multithreaded CPU – in order to benefit from thread-level parallelism. As we will see, several of the programs do in fact not have very much exploitable MLP.

3. PREDICTING MISSPECULATIONS

The misspeculation prediction technique attempts to selectively disable speculation whenever a thread causes misspeculations. Every time there is a dependence violation, the call tree is analyzed in order to find a confluence point where, if a new speculative thread is not spawned, the dependence violation will disappear. Once the confluence point is identified, we propose to use history-based prediction in order to avoid expected misspeculations when the same situation occurs again.

The main advantage of this method is that all misspeculating threads are targeted while successful threads are left alone, which ought to ensure a significant reduction in misspeculations and therefore low overhead. A drawback is that threads which misspeculate early and later contribute with useful parallelism are also affected; we do not search for the best tradeoff for maximum speedup.

We describe the method in Section 3.1. The impact on speedup and overhead is then investigated in Section 3.2.

3.1 Algorithm & Implementation

A first concern is which call instruction to classify as non-parallel after a violation. In fact, there are several possibilities to select module(s) as non-parallel in order to get rid of the misspeculation. In Figure 4, there is a dependence violation between a store in thread T1 and a load in T4. In order to avoid the violation, we must make sure the store is executed before the load. To achieve this, one or several of the confluence points (*A*, *B*, and *C* in the figure) that define the relative position of these instructions must be selected.

Choosing either *A* or *B* alone might do the trick, since both will delay the execution of the load. Which one is better to select will depend on the situation. Intuitively, a good heuristic could be to select *A*, the common ancestor for the threads involved in the violation, as being more likely to delay the load sufficiently. The advantage of *B*, however, is that it is the confluence point where the misspeculating thread was created; thus it will be easy to find the right call instruction to mark non-parallel in conjunction with the

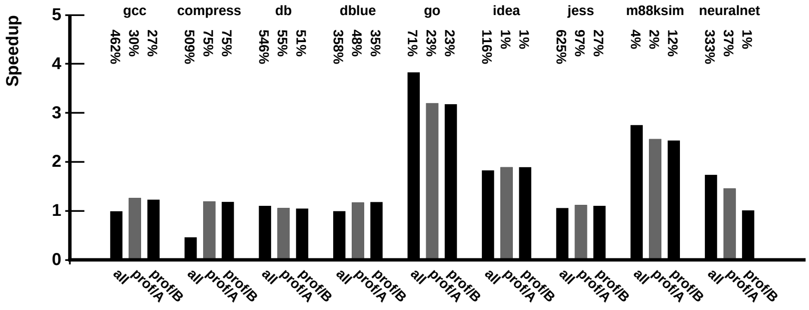


Figure 5: Profiling results for disabling speculation based on misspeculations, type A and B.

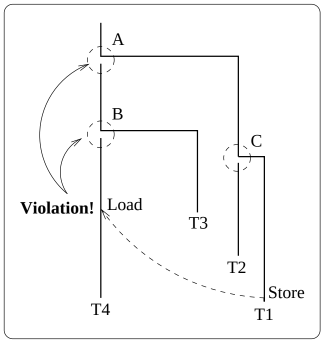


Figure 4: Identifying calls to classify as non-parallel.

roll-back operation. Only *A* and *C* together are certain to remove the violation, serializing all code from the common ancestor to the problematic store instruction. However, this is undesirable for several reasons: first, we do not want to remove more parallelism than necessary to avoid the violation; second, finding and inserting multiple calls in the prediction table would take more time; and finally, if the first attempt failed, the predictor will add another non-parallel prediction in order to get rid of the violation if the situation repeats. We will experiment with both the closest fork (called type *B*), and the common ancestor (type *A*) predictors.

Implementation of this technique requires that the roll-back handler is augmented to find type *A* or *B* confluence points based on the knowledge that T1 and T4 contains the conflicting instruction pair. The relevant confluence points can be found using the list of threads active in the speculation system; this list is already accessed by the roll-back mechanism in the course of squashing, so the overhead should be small compared to the existing mechanism. A prediction table is also needed; it will be accessed at module calls and updated during roll-backs if needed. The prediction table will be discussed in the next section.

3.2 Experimental Results

We begin with profiling experiments. The workload is run several times, each time all type *A* or *B* calls that cause a misspeculation are marked non-parallel when re-executing the same workload. This process is repeated until we no longer have any misspeculations or a maximum of five iterations, lest we spend too much time finding and removing the last few misspeculations. While the profiling method is approximate, it will still give a fair estimate of the potential for using misspeculation prediction to reduce the speculation overhead.

In Figure 5, the leftmost bar for each application, marked *all*, is the naive implementation of running all modules speculatively. The next two bars, *prof/A* and *prof/B*, show results for the profiling runs for type A and B confluence points respectively. For each application, the height of the bars indicate speedup on the speculative CMP compared to sequential execution. The numbers on top of the speedup bars show the total overhead as defined in Section 2.2.

The average overhead for indiscriminate speculation is as high as 336%. Even programs with decent speedup can have high overhead, for instance, Neuralnet with a speedup of almost 2 and 333% overhead. This is possible since we have an 8-way machine and therefore capacity to get useful work done even if some threads are repeatedly squashed and re-executed.

As we expected, the total overhead is greatly reduced. The average overhead for type A is 41% and for type B 28%; in no case does the overhead exceed 100% of the serial execution time. The lower average overhead for type B can be attributed to lower overhead figures for Jess and Neuralnet. However, this is at the expense of worse speedup, especially for Neuralnet which is almost serialized with *prof/B*. In addition, with type B there are actually more threads squashed than in indiscriminate speculation for M88ksim, resulting in the higher overhead figure. For these reasons type A, highlighted in grey, seems to be the better choice. Speedup improves for five programs, but is reduced in four cases (although marginally for Db) compared to running everything speculatively.

In summary, by removing misspeculating threads with profiling, we can bring down the overhead from an average of 336% to an average of 41% or 28% while at the same time improving speedup for about half of the applications. While this is encouraging, in the next section we will investigate how well a predictor can exploit this potential. However, we do run the risk of disabling speculation even where we do have useful parallelism. This happens in a few of the benchmarks, notably Go, M88ksim, and Neuralnet. This problem, and a potential remedy, will be discussed in Section 5.

4. DESIGN SPACE FOR MISSPECULATION PREDICTORS

The misspeculation prediction technique shows promise in bringing down the overhead in module-level speculation. In this section we will make a thorough investigation of the design space. We begin with a discussion of the design space before analyzing the performance of a number of designs.

4.1 Predictors & Implementation

Once unwanted threads are identified, we use a prediction table to store information about the misspeculations. The *indexing*

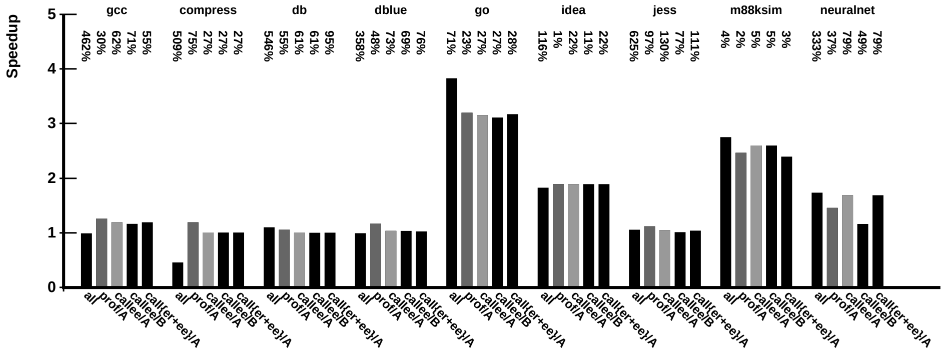


Figure 6: Comparison of misspeculation prediction policies.

method determines how future events are matched and identified as probable misspeculations. We want to catch future instances where the same module is called and we expect yet another misspeculation. However, the same module can be called from multiple places in the code. This means an entry in the prediction table could cover everything from only one of those calls to all of them, depending on how we chose the prediction table index. Having a single entry cover multiple call places, for instance, saves space in the prediction table, and the warm-up time will be shorter. On the other hand, prediction accuracy might suffer.

Some interesting options are:

- Per-call: The prediction table is indexed with the call instruction address; i.e. the table entry only covers a single call instruction in the application.
- Caller/Callee: A concatenation of the module IDs for caller and callee modules: the scope is expanded to cover similar situations in the same module (repeated calls to the same function).
- Callee only: Index by module ID, the same table entry is used regardless of where the module was called from.

The module ID could be any identifier unique for the module, such as a sequence number or the address of the first instruction. We have not yet had the opportunity to evaluate the per-call option, but the latter two are evaluated and compared in the next section.

The next question is which *predictor* to use. We begin with a simple last-outcome predictor, which will disable speculation on a module as soon as a violation has occurred. In order to avoid making a decision based on an exceptional case, the last-outcome predictor can be enhanced to an *n*-bit saturating counter type predictor. We have run experiments with last-outcome and 2-bit predictors. For the 2-bit predictor, speculation is disabled when the high bit is set, i.e. after two consecutive misspeculations.

A disadvantage of our misspeculation prediction technique is that we lack the ability to re-evaluate a no-speculate prediction. Once speculation is disabled, we can no longer detect if circumstances change since future invocations will be run sequentially. The last point in our design space is *prediction duration*. We either let the prediction be permanent, or we let it time out with some interval in order to make a reevaluation. We will investigate having the prediction time out and reset to zero after it has been accessed *k* times.

The implementation issues are the same as those described in Section 3.1, however, we did not discuss the prediction table. The table should be shared among the cores in the CMP. It could be stored in memory, where it is automatically shared between the processors. Since the table is updated only when a prediction changes – at first misspeculation or timeout – most accesses will be read-only, which should help keeping sharing overhead due to invalidations relatively low. Preferably the table should not take up too much space. As is shown in the rightmost column in Table 2, there are only up to a few hundred modules in the applications. Thus, if we store predictions per module ID, the table should not need to contain more than a few hundred entries to avoid having several modules map to the same slot. The other options will require somewhat larger tables to avoid interference. In most of the simulations we assume an infinite prediction table, which is not unreasonable due to the small number of modules; however, we will also run some experiments with finite table size. The predictions use one or two bits each, plus an expiration counter if we use timeout. For the ranges of interest, a six-bit timeout counter would suffice.

4.2 Experimental Results

4.2.1 *Impact of prediction table indexing method*

In Figure 6, we compare three ways to store a prediction, The *Callee/A* and *Callee/B* bars show speedup when storing predictions based on callee module ID, and choosing the module of type *A* or *B* respectively. The rightmost bar, labeled *Caller+Callee/A*, instead uses a concatenation of the caller and callee module IDs as index for the prediction table. The *all* and *prof/A* results are carried over from the previous section for comparison. In these simulations, a last-outcome predictor is used.

We can see that the methods perform similarly in most cases, both with respect to overhead and speedup. Only for M88ksim does the *Caller+Callee/A* give a lower speedup than the others, and the same is true for Neuralnet and *Callee/B*. We can conclude that the increased resolution of caller+callee IDs does not improve the result. Since the simpler module ID indexing method is more space-efficient, we rule out the caller+callee option. *Callee/A* seems to consistently yield the best results, which conforms with the profiling results.

The predictor generally runs more modules speculatively than the profiler, with Compress being the exception. Not all misspeculating modules are correctly predicted as such; hence the predictor shows somewhat higher overhead figures, and in most cases lower speedup. However, for M88ksim and Neuralnet the speedup is better with more speculation; some parallelism is lost when disabling misspeculating modules in these applications.

4.2.2 *Impact of choice of predictor*

Based on the choice of *Callee/A*, we proceed to examine three different predictors, the last-outcome, a 2-bit predictor, and a 2-bit

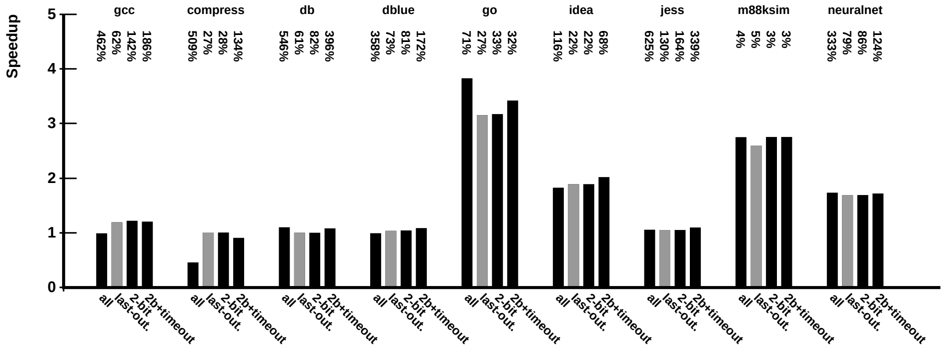


Figure 7: Performance of the last-outcome, 2-bit and 2-bit + timeout predictors.

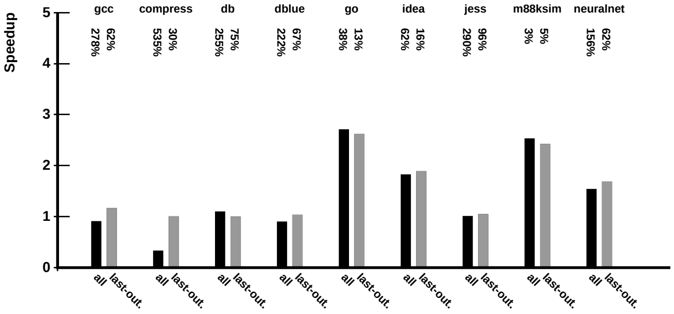


Figure 8: Performance of last-outcome misspeculation predictor with a 4-way CMP.

predictor with timeout. The results are shown in Figure 7. Timeout is set so that a prediction expires after it has been accessed 20 times. We investigated different timeouts in the range of 10-100 accesses, with 20 showing the best overall result.

It is clear that last-outcome and 2-bit predictors perform virtually the same, only for M88ksim is there a slight improvement from using the 2-bit predictor. However, because the 2-bit predictor takes longer before reaching the decision to disable speculation, the number of misspeculations and consequently the total overhead is generally somewhat larger; significantly larger in the case of Gcc. With timeout added, the difference in overhead is even more pronounced. Some programs benefit from the timeout, namely Go, Idea, and Db, but the overhead of Db also increases from 82% to 396% with the timeout enabled. The average overhead is 54% with a last-outcome predictor, not unreasonably higher than the 41% reported by the profiling run. The 2-bit predictor has a slightly higher 64% average overhead, and with timeout we get a significantly higher 161%.

4.2.3 Impact of number of processors and speculation overhead

In order to make sure the last-outcome misspeculation predictor is beneficial with a range of hardware organizations, we reran the experiment on a smaller 4-way machine. The results for the 4-way machine are shown in Figure 8. In Figure 9 we once again use an 8-way machine but with lower 10- or 50-cycle speculation overheads.

As expected, the total overhead is lower on the 4-way machine than the 8-way. Since fewer processors are available, we simply cannot start as many threads simultaneously, and consequently less work is squashed due to violations. However, there is still an average 204% total overhead when starting speculative threads for all continuations, compared to an average 47% when employing our technique. The impact on speedup is very similar to what we can see for the eight-way configuration.

The second variation is the speculation overhead, since we do not know exactly how much overhead will be imposed in a real implementation. The technique is less likely to yield good results on machines with low speculation overheads; therefore, we reran the experiments with 10- and 50-cycle overheads. As is evident in Figure 9, the misspeculation predictor still produces good results. With 50-cycle overheads, the average total overhead goes down from 274% to 53% with our technique. With 10-cycle overheads, the average decreases from 230% to 51%. The total overhead does not decrease as much as one might think with lower speculation overheads. This is due to the fact that the number of misspeculations increases when threads are started in a faster pace. With lower speculation overheads, however, speedup is less affected by misspeculations. Hence, we can see that speedup suffers somewhat for all programs with misspeculation prediction enabled when overheads are as low as 10 cycles.

4.2.4 Impact of finite size prediction table

All previous results are with an unlimited prediction table, i.e. there are never interference between two modules with a different ID. Since the number of different modules is not huge (see Table 2) one can expect that a relatively small predictor table will be sufficient for our misspeculation predictor. Figure 10 shows results with finite prediction tables of 1024 and 256 entries. Each entry is a single bit containing the last-outcome predictor. The leftmost bar shows the unlimited table used in previous Figures, while the other two bars show results with 1024-entry and 256-entry tables. The tables are indexed with 10 and 8 bits from a 32-bit module-ID respectively. We can see that the performance is close to that of

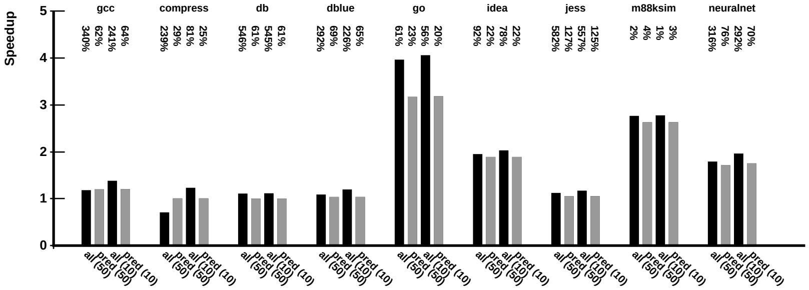


Figure 9: Performance of last-outcome misspeculation predictor with 10- and 50-cycle overheads.

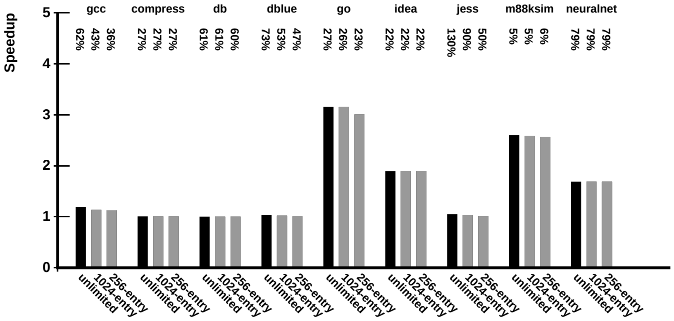


Figure 10: Performance with realistic 256- and 1024-entry prediction tables.

the unlimited predictor. When there is some interference, such as for Gcc, the result is that a few modules that should have been run speculatively are instead run sequentially. The overhead goes down somewhat, but at the expense of lower speedup. With a 1024-entry table, four of the nine benchmarks perform identical to the unlimited predictor, and the impact on the remaining five is small.

In summary, the last-outcome predictor with the *Callee/A* table, highlighted in grey in Figure 7, seems to be the best choice, yielding a slight speedup improvement on four programs, and the same speedup on two, but with a significantly lower 54% average overhead. Even if the architecture in terms of number of processors or size of overhead changes, the gain achieved with misspeculation prediction remains significant. In addition, the gain can be achieved with a relatively small prediction table.

However, a couple of the programs, Go and m88ksim, works better without the technique enabled at all. The overhead is small to begin with, and using misspeculation prediction removes useful parallelism and increases the execution time. In the next section we will look at a possible solution for that problem.

5. SELECTIVE USE OF MISSPECULATION PREDICTION

The results from the previous sections show that misspeculation prediction is an efficient way to reduce the misspeculation overhead while achieving the same or slightly higher speedups as running everything speculatively. However, Go and M88ksim did not benefit from the technique. On the contrary, their speedups are negatively affected. These two programs show good speedups and low overhead without applying a misspeculation reducing technique – there are few misspeculations in these applications to begin with.

In this section, we attempt to add a safeguard which will make sure that misspeculation prediction is not applied to programs which do better without. The reasoning is simple – if there are many misspeculations the technique is enabled and the predictor used when deciding if a new thread should be created or not; if misspeculations are relatively few, the prediction table is not used.

5.1 Algorithm & Implementation

In order to get a metric of how prevalent misspeculations are in a program, we maintain two global counters: a squash counter is increased every time a thread is squashed, and a thread-start counter is increased every time a new thread is started. The ratio squash/thread-starts will, at any point in the execution, be a value between 0 and 1 which shows the fraction of the started threads that have been squashed. If there are many misspeculations, the number goes up; if speculation is successful, the number goes down.

The idea is to have misspeculation prediction, as described in the previous section, with predictors being updated throughout the execution time. However, the predictors are only used in the decision of whether to create a new thread or not if the squash/thread-start number is above a threshold. That way, the use of misspeculation prediction will be automatically enabled and disabled as needed during the execution of the program.

Implementation is simple, only two global counters, increased by the roll-back and thread-start handlers respectively, need to be added.

5.2 Experimental Results

In order to find out if there indeed is a useful threshold we ran simulations with thresholds in increments of 0.05. In Figure 11, the interesting range of threshold values is shown. Some of the applications are well on either side of the threshold. Misspeculation prediction is always enabled for Db, Idea, and Neuralnet, and always disabled for M88ksim, in this range. For Gcc, Dblue, and Jess we see how the overhead steadily increases as the threshold is

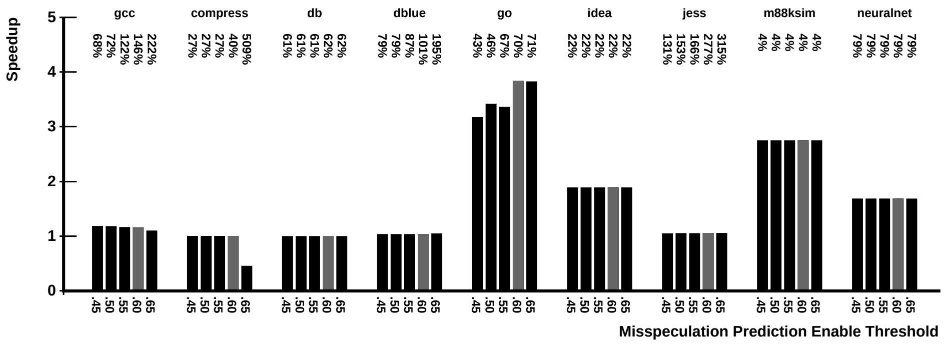


Figure 11: Threshold for misspeculation prediction.

increased to allow more misspeculations before the predictors are used, but there is yet no change in speedup.

The sensitive applications are Go and Compress. Note that when the threshold is set to 0.65 the speedup for Compress goes down sharply. In fact, we get a large slowdown, due to the fact that misspeculation prediction is permanently disabled. For Go, the opposite is true, when the threshold is 0.55 or lower, misspeculation prediction is active and removes some useful parallelism. Only a threshold of around 0.6 is fine for all the applications.

With a threshold of 0.6, the average overhead is 89%, up from 54% when using misspeculation prediction without threshold, but with overall better speedup. However, the best threshold is within a rather narrow range, so the potential drawback is that this threshold might not always yield the best results over a larger number of applications. Keeping the threshold as low as possible will at least make sure the overhead is brought down and at the same time decrease the likelihood of suffering from slowdown.

6. RELATED WORK

An alternative method used to prevent misspeculations is to learn about cross-thread dependences and stall the dependent load until the dependency is resolved. This has been investigated in the context of Multiscalar processors [11], speculative chip multiprocessors [16] and larger DSM machines [4] with some success. The Superthreaded architecture [18] does not speculate on data dependences at all; instead, inter-thread data dependences are always solved with synchronization, at the price of serialization of all store address calculations.

For the Multiscalar processor [11], load-store pairs that are predicted to cause a violation are inserted in a synchronization table. Using this table, dependent loads are stalled until the corresponding store has completed and the value can be forwarded. The technique described by Steffan et al. [16] is slightly different; a list of violating loads is maintained and when a load that appears on the list is encountered, the thread is stalled until it becomes non-speculative. Finally, the technique by Cintra and Torrellas [4] is similar to the one by Steffan: however, they use two levels of stalling: Stall&Release, where the load is stalled until the first writer thread has committed, and if this fails Stall&Wait, which stalls the thread with the load until it is non-speculative. In Hammond et al. [6] they use a simpler synchronization method; the compiler may insert explicit synchronization into the code in the form of a busy-wait loop that reads a lock variable and a store that writes the same lock. Our technique differs from these since we try to avoid creating threads which will misspeculate in the first place; which means that we also avoid the thread-start overhead.

It is mentioned that Hydra uses techniques such as thread timers, stall timers, and violation counters to disable speculation for non-parallel threads and thus decrease overhead [6]. However, neither the implementation nor the performance of these techniques have been reported; thus it is unclear how they relate to our technique.

7. CONCLUSIONS

When aggressively spawning speculative threads at all module invocations, the execution is dominated by overhead. In our benchmark applications, we have found that the average overhead is three times as big as the useful work with indiscriminate speculation.

The technique presented in this paper is aimed at bringing down the overhead in order to save processing and communication resources, as well as reduce the extra energy required for thread-level speculation. The technique can be integrated in the speculation runtime system and does not require recompilation of the programs.

Our experimental findings are the following:

- The overhead can be reduced with a factor of six using a last-outcome misspeculation predictor for each module. The speculation system decides whether or not to start a new thread when a module is called based on this prediction. However, the speedup is adversely affected for some applications.

- Our technique is shown to work well for a number of chip multiprocessor architectures with varying number of cores and size of speculation overhead. In addition, the technique is shown to work well with a small (in the range of a few hundred entries) shared prediction table.

- When adding a mechanism for dynamically enabling and disabling misspeculation prediction based on whether the ratio of misspeculations to new threads is above a certain threshold (0.6 was found to be the best threshold for our applications) the average overhead is reduced a factor of four, but with equal or better speedup than indiscriminate speculation for all the benchmark applications.

Overall, this study shows that it is possible to exploit most of the inherent speculative module-level parallelism while removing most of the overhead associated with indiscriminate speculation.

8. ACKNOWLEDGMENTS

This research has been supported by the Swedish Foundation for Strategic Research under the ARTES/PAMP program.

9. REFERENCES

[1] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*, pages 282–293. ACM Press, June 2000.

[2] M. K. Chen and K. Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 176–184. IEEE Computer Society, Oct. 1998.

[3] Y. Chen, R. Sendag, and D. J. Lilja. Using incorrect speculation to prefetch data in a concurrent multithreaded processor. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '03)*, page 76. IEEE Computer Society, Apr. 2003.

[4] M. Cintra and J. Torrellas. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *Proceedings of the Eight International Symposium on High-Performance Computer Architecture (HPCA '02)*, pages 43–54. IEEE Computer Society, Feb. 2002.

[5] L. Codrescu and D. S. Wills. Architecture of the atlas chip-multiprocessor: Dynamically parallelizing irregular applications. In *Proceedings of the 1999 International Conference on Computer Design (ICCD '99)*, pages 428–435. IEEE Computer Society, Oct. 1999.

[6] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII '98)*, pages 58–69. ACM Press, Oct. 1998.

[7] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.

[8] P. S. Magnusson, F. Larsson, A. Moestedt, B. Werner, F. Dahlgren, M. Karlsson, F. Lundholm, J. Nilsson, P. Stenström, and H. Grahn. SimICS/sun4m: A virtual workstation. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 119–130. USENIX Association, June 1998.

[9] P. Marcuello and A. Gonzalez. Clustered speculative multithreaded processors. In *Proceedings of the 1999 International Conference on Supercomputing (ICS '99)*, pages 365–372. ACM Press, June 1999.

[10] P. Marcuello and A. Gonzalez. A quantitative assessment of thread-level speculation techniques. In *Proceedings of the 14th International Conference on Parallel and Distributed Processing Symposium (IPDPS '00)*, pages 595–604. IEEE Computer Society, May 2000.

[11] A. Moshovos and G. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97)*, pages 181–193. IEEE Computer Society, May 1997.

[12] C.-L. Ooi, S. W. Kim, I. Park, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar. Multiplex: unifying conventional and speculative thread-level parallelism on a chip multiprocessor. In *International Conference on Supercomputing (ICS '01)*, pages 368–380. ACM Press, June 2001.

[13] J. Oplinger and M. S. Lam. Enhancing software reliability with speculative threads. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '02)*, pages 184–196. ACM Press, Oct. 2002.

[14] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*, pages 303–313. IEEE Computer Society, Oct. 1999.

[15] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, pages 414–425. ACM Press, June 1995.

[16] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *Proceedings of the Eight International Symposium on High-Performance Computer Architecture (HPCA '02)*, page 65. IEEE Computer Society, Feb. 2002.

[17] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA '98)*, pages 2–13. IEEE Computer Society, Feb. 1998.

[18] J.-Y. Tsai and P.-C. Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 35–46. IEEE Computer Society, Oct. 1996.

[19] F. Warg and P. Stenström. Limits on speculative module-level parallelism in imperative and object-oriented programs on CMP platforms. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '01)*, pages 221–230. IEEE Computer Society, Sept. 2001.

[20] F. Warg and P. Stenström. Improving speculative thread-level parallelism through module run-length prediction. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '03)*, page 12. IEEE Computer Society, Apr. 2003.