Improving Speculative Thread-Level Parallelism Through Module Run-Length Prediction

Fredrik Warg and Per Stenström
Department of Computer Engineering
Chalmers University of Technology
{*warg,pers*}@*ce.chalmers.se*

Abstract

Exploiting speculative thread-level parallelism across modules, e.g., methods, procedures, or functions, have shown promise. However, misspeculations and task creation overhead are known to adversely impact the speedup if too many small modules are executed speculatively. Our approach to reduce the impact of these overheads is to disable speculation on modules with a run-length below a certain threshold.

We first confirm that if speculation is disabled on modules with an execution time – or run-length – shorter than a threshold comparable to the overheads, we obtain nearly as high speedup as if the overhead was zero. For example, if the overhead is 200 cycles and the run-length threshold is 500 cycles, six out of the nine applications we ran enjoyed nearly as high speedup as were the overhead zero. We then propose a mechanism by which the run-length can be predicted at run-time based on previous invocations of the module. This simple predictor achieves an accuracy between 83% and 99%. Finally, our run-length predictor is shown to improve the efficiency of module-level speculation by preventing modules with short run-lengths from occupying precious processing resources.

1 Introduction

Speculative thread-level parallelism (STLP) is an attempt to extract parallelism at a coarser level than instruction-level parallelism, by automatically splitting up programs into threads that are executed in parallel on multiple processor cores. With this approach, threads do not need to be provably data independent; instead, the STLP machine will check for dependences at run-time, and if necessary roll back and re-execute threads when data dependence violations occur. Several implementation proposals for STLP machines exist, often in the form of chip-multiprocessors [3, 6, 8, 10, 13, 15, 16].

To get good utilization of an STLP machine, however, we need methods for efficiently determining when and how to create new threads. One approach is to use procedure, function or method calls (collectively refer to as *modules* in this paper) as the point to spawn threads [2, 6, 7, 11, 12]. At a module invocation, a new thread that continues execution after the call instruction is created; the old thread executes the module and then terminates. The advantage of module-level parallelism is straight-forward identification of threads and avoidance of the control-dependency problem that plague e.g. exploitation of loop-level parallelism. Our focus is to explore the feasibility of speculative module-level parallelism in the context of chip-multiprocessors with hardware STLP support.

In a previous study [17] we found that while programs from SPECint95 and SPEC JVM98 can enjoy a speedup ranging from two to six on an eight-processor CMP; achieving this speedup is mainly limited by the overhead associated with thread management and misspeculations.

Thread creation/termination, roll-backs, and context switches are all associated with significant overhead. If the overhead is significant in comparison with the module execution time – or run-length – the contribution to the overall speedup is small. Hammond et al. [6] found threads of size 300-3000 instructions suitable if overheads are in the range 10-100 cycles. Consequently, using the run-length of a module as a key criterion for selecting which modules to speculate on appears to be a promising way to reduce thread management overhead. The potential of this technique is explored in this paper.

We first investigate how much speedup we can gain if we only speculate on modules with a run-length greater than a certain threshold. Based on nine Java and SPECint95 applications, it is possible to eliminate most of the impact of overhead in the range of 100-500 cycles on speedup by only speculating on modules whose run-length is above a certain threshold, typically around 500, assuming perfect a priori knowledge of the run-length.

We then introduce the design of a module run-length predictor that, based on the previous run-length of the mod-

0-7695-1926-1/03/$17.00 (C) 2003 IEEE

ule, will predict if future invocations of the module will exceed the threshold or not. This predictor is shown to behave very close to the off-line omniscient predictor with a prediction accuracy between 83% and 99%. We demonstrate that such a predictor can wipe out almost all of the impact of thread-management overhead on the overall speedup of the applications on an 8-way chip-multiprocessor with support for STLP. As opposed to related off-line techniques such as compiler inlining, our method can be used for run-time speculative parallelization of sequential binaries.

Finally, we apply our run-length predictor to machines with a limited number of simultaneous speculative states (or maximum number of live threads). Two benchmarks benefit much from the run-length predictor, which reduces the number of threads created.

In Section 2 we explain the execution model for module-level parallelism, and present our simulation environment. Then, in Section 3, we look at module run-length thresholds and their impact on overhead penalty. Section 4 introduces the run-length predictor, and in Section 5 its performance is compared to that of a perfect predictor. Section 6 discusses the problem with speculative state, and finally, we conclude the paper in Section 7.

2 Experimental Methodology

In this section, we begin by briefly explaining the basic execution model behind speculative module-level parallelism. Then, we describe our simulation environment, and present the benchmarks we have used.

2.1 Execution Model

In speculative module-level parallelism, module calls are the points where new threads are potentially spawned. At a call instruction, a new thread will execute the code after the call, the module continuation, while the old thread executes the called module. In order to respect sequential semantics and correctly perform the data dependence checking, we need to keep track of the sequential order of all threads. A new thread will be more speculative than its parent, and retain the speculative order of the parent with respect to all other threads.

A graphical representation of new threads being created is shown in Figure 1. To the left in the figure is a C program with function calls, and to the right a call tree representation of the same code run with the module continuations as separate threads.

Data dependences between threads can cause speculation to fail. A flow dependency occurs when a less speculative thread computes a value used by a more speculative thread. If the less speculative thread writes the value before the more speculative reads it, the correct value can be forwarded, but otherwise the more speculative thread will have already used an incorrect value. If that happens, the STLP system restores the execution to a safe state, which includes rolling back execution of the violating thread and squashing any thread started after the roll-back point. Most proposed systems solve this by storing speculative values either in the cache system or in special (hardware- or software-managed) buffers. The speculative state can then be committed when the thread it belongs to is the head (non-speculative) thread, or flushed if a violation occurs.

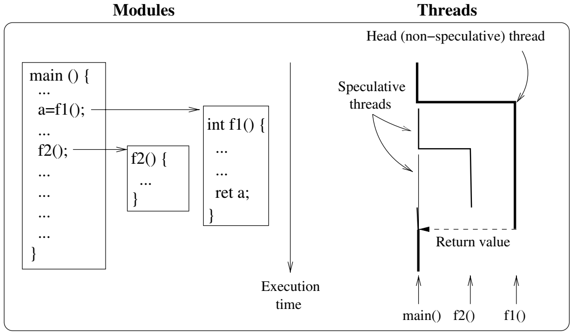

Figure 1. Execution model. New speculative threads are spawned for module continuations.

2.2 Simulated STLP Machine

Since this study focuses on the impact of thread management overhead on the inherent module-level parallelism, the philosophy behind our machine model is to only factor in such overheads and disregard others, for example inefficiences in the memory system. We consider an STLP machine with eight processors as our results in [17] indicated that eight processors are enough to exploit almost all module-level parallelism in our benchmark applications. Additionally, we have also made the following machine model assumptions:

- All communication between threads as well as memory accesses only takes one cycle and the processor cores issue one instruction per cycle in order.
- Fixed-length overhead is imposed on thread-starts, roll-backs and context switches; in the simulations we use values between 100 and 500 cycles.
- Threads can be preempted. If a new thread is less speculative than an already running thread and there are no free processors, the running thread will be preempted and resume at a later time. The intuition is that flow dependences are more likely to be resolved with forwarding if less speculative threads are favored over more speculative ones.
- A thread can roll back to the very instruction that caused the dependency (perfect rollback); we do not have to re-execute the whole thread. Threads started

Table 1. Benchmark applications.

Name	*Origin*	*Description*	*#Instructions*	*#Modules*	*Avg. instr./mod.*	*#Modules*
			(dynamic)	*(dynamic)*	*(dynamic)*	*(static)*
gcc	SPECint95	GNU C Compiler 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	Plays 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

by the violating thread after the rollback are always squashed, however.

- The speculation system can resolve anti- and output-dependences as well as handle forwarding of values.
- We use *perfect value prediction*, and *realistic value prediction* models. With perfect value prediction we assume all dependences are resolved and no roll-backs are necessary. The realistic model uses simple stride prediction for return values and no prediction at all for memory references. Unless otherwise stated, results are for the realistic model.

Commit and dependence checking can be done with none or very low overhead, therefore we do not model any overhead for these events. Commit might take a little time, but is in any event less crucial than the three mentioned above, since it only happens once per thread, at commit time, and will not be affected by roll-backs or squashes.

2.3 Simulation Setup

Our simulation results are produced with a trace-driven simulator used in a previous study [17]. The programs are first run sequentially on the system-level instruction set simulator Simics [9]. In the generated trace, all events such as module invocations, returns, and loads and stores are annotated with a time-stamp generated by the virtual timer. The STLP machine model is then driven by the trace and when an event is encountered, the STLP machine creates new threads, does dependence checking, roll-back etc according to the execution model in the previous section.

The benchmarks were compiled with GCC 2.95.2 with full optimizations, and run on Linux. The whole system runs on top of Simics, which does call-backs to our trace-generator when encountering one of the events mentioned above. This way we will not introduce any overhead in the simulated application. Simics simulates a single-issue in-order SPARC v8. The memory system is perfect (one-cycle access).

Using a realistic (out-of-order) processor core, memory hierarchy, and communication mechanisms would introduce a number of additional effects, like inter-thread communication latencies, bus contention, speculative state management, memory-access latencies etc. While it eventually is important to study their impact, we opted for studying the impact of thread management in isolation and not factoring in these effects.

2.4 Benchmarks

We have used nine benchmarks, four from SPECint95, two from SPECjvm98, two from jBYTEmark, and one constraint solver written at Sun. The choice was made for the sake of comparison; many of these programs have been used in related studies [7, 11, 12] and are also the same programs used in our own previous study on module-level parallelism [17]. We only consider integer programs that have been shown hard to parallelize with static methods such as parallelizing compilers.

Table 1 summarizes the benchmark applications.

3 Potential of Run-Length Thresholds

In order to demonstrate the impact of thread management overheads on the potential speedup of speculative module-level parallelism, we ran simulations with thread-start, roll-back, and context-switch overheads. In Hydra [6], speculation events are handled by a speculation coprocessor where control routines of typically 50-100 instructions are executed for each event. While these overheads are useful as reference points, it is unclear how many cycles of overhead we will see in future STLP machine implementations. Therefore, we use overheads ranging between zero and 500 cycles per event in order to study the sensitivity of the overhead impact on speculative module-level parallelism.

In Figure 2 the speedup of our nine applications for different overheads is shown. The upper graph shows simulations with the perfect value prediction model, and the lower graph with realistic value prediction. As expected, for overheads of 100 cycles, the speedup is already severely hampered, especially under the realistic model where roll-backs and thread restarts kick in. Moreover, with a 500-

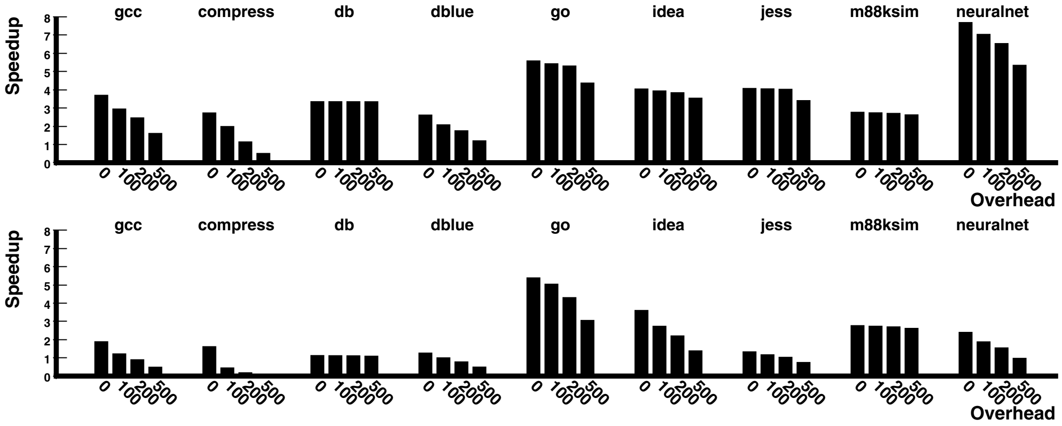

Figure 2. Speedup with thread-management overhead 0-500. The graphs show results with perfect (upper) and realistic (lower) value prediction models.

cycle overhead, speedup is more than halved for most applications. M88ksim is less affected by roll-backs, and thus experiences less events causing overhead. Compress, on the other hand, which largely consists of very small modules, already suffers from a slowdown at 100-cycle overheads.

In order to better amortize the overhead costs over the useful execution, we want to avoid creating new threads which do not contribute to the speedup or worse, tie up machine resources with little gain. We do this by applying a threshold on the module run-length. If the run-length exceeds the threshold, a new thread is created for the module continuation. If not, the overhead is expected to negate any positive effect of the gain in parallelism, so the code is run sequentially.

We define module run-length as the time between the call and return of a module. As shown in Figure 3, this time will include the run-time for child modules run sequentially, but exclude run-time for child modules when new threads are created. Overhead is not included. The module run-length gives a measure of how much useful overlapping of the execution a new thread is expected to yield.

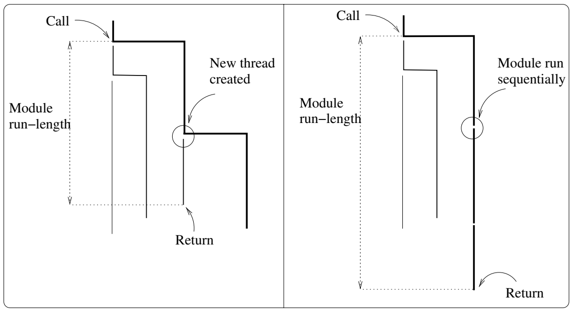

Figure 3. Module run-length calculation.

Since we use trace-driven simulation, we can precompute the dynamic run-lengths from the execution traces and use this a priori knowledge when applying different run-length thresholds.

Figure 4 shows the speedup for our benchmark applications with thresholds between 0 and 10000 cycles. We show full speedup graphs for Gcc, Go, and NeuralNet, and abridged versions (only three thresholds) for the remaining applications. Gcc and NeuralNet were chosen as good examples of the usefulness of module run-length thresholds, whereas Go is included to show some unusual behavior. In the full graphs, each line represents a different amount of overhead. The vertical axis shows speedup and the horizontal axis different thresholds. Note that the vertical scales are different for the applications.

In the bar graph, we depict different overheads with shaded sections. The whole bar shows speedup for zero overhead. Then, progressively darker sections show speedup with 100, 200, and 500 (black section) cycle overheads respectively. For example, speedup for compress without a run-length threshold (or threshold=0) is: with zero overhead 1.64, for OH=100 it is 0.47, for OH=200 only 0.21, and for OH=500 it is 0.1.

We get a speedup improvement on all applications except Db and M88ksim. This is expected as Db and M88ksim have a larger portion of long modules and the impact of overhead is small. Jess and Deltablue show improvements, and a small positive speedup; without module run-length threshold they suffer a slowdown. Gcc shows potential with ideal value prediction but suffers badly from misspeculations, which a run-length threshold does not solve. Compress hardly has any parallelism with a threshold of 100 or above. The run-length predictor effectively nullifies the overhead so that it runs at least sequentially with no overhead. Go has a lower best threshold than the other applications. The best result is achieved for a threshold of around

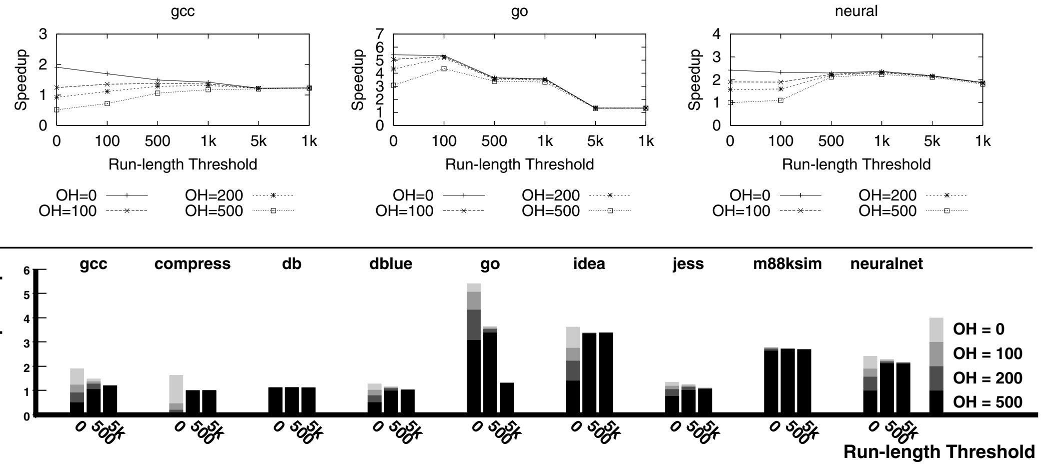

Figure 4. Speedup with module run-length thresholds between 0 and 10000 cycles.

100; at 500 the speedup is down again due to a lack of parallelism.

Overall, for six out of nine applications, the speedup at an overhead of 200 is very close to the speedup without overhead when a good run-length threshold is used, and none of the programs suffer from slowdown.

4 Module Run-Length Prediction

In the previous section we saw that creating new speculative threads only when the module run-length exceeds a threshold can help alleviate the impact of thread-management overheads. However, the decision to create a new thread needs to be done when the module is called, and we cannot know the run-time until it has completed execution. To overcome this problem, we make use of a technique common in computer architecture: history-based prediction. It is reasonable to assume that there is a correlation between the run-length of one invocation of a module to the next.

Our predictor works like this:

- Each module in the application has its own predictor associated with it. The predictor uses a single bit which designates whether run-time was above or below the run-length threshold for the most recent completed execution of the module.
- The module run-length is measured every time the module is called. When it completes (reaches return), the measured run-length is compared to the threshold. If it exceeds the threshold, a '1' is stored in the predictor bit, otherwise, a '0' is stored.
- When execution reaches a module call, the prediction bit is checked. If the bit is '1', a new thread is created for the continuation, otherwise the module is run sequentially.
- All prediction bits are initialized to '1', so on the first invocation a new thread will always be created.
- We have assumed zero-latency for the prediction mechanism in our simulations.

Note that the run-length is measured regardless whether a new thread is created or not; otherwise a module that has once been marked '0' would no longer be updated, and the prediction could never change. Since the result of prediction changes further down the call tree can propagate to parent modules, it is especially important that predictor changes can go both ways; it might take a few invocations before the predictor reaches steady state.

The possible advantages of measuring module sizes dynamically instead of doing static analysis is that the length may be hard or impossible to determine statically. In addition, a dynamic predictor can automatically adjust to hardware dependent parameters such as communication and memory latency. It is likely, however, that a combination could be useful. For instance, very small modules whose length can be determined statically could be removed from being considered for speculation, in order to minimize overhead from the run-length measurements.

In order to implement run-length prediction, methods for measuring the run-length as well as a structure for storing history-bits and temporary cycle counts is needed. Storage should be shared among the processors in the CMP in order to support preemption and shared prediction bits.

The storage could be implemented as a dedicated hardware structure, or in order to avoid extra hardware, in the memory hierarchy. As we can see in Table 1, the number of unique modules is at most a few hundred, so the structure need not be very large.

Most existing processors have performance counters, including a cycle-counter, which could be used for a software implementation of run-length prediction. Measuring

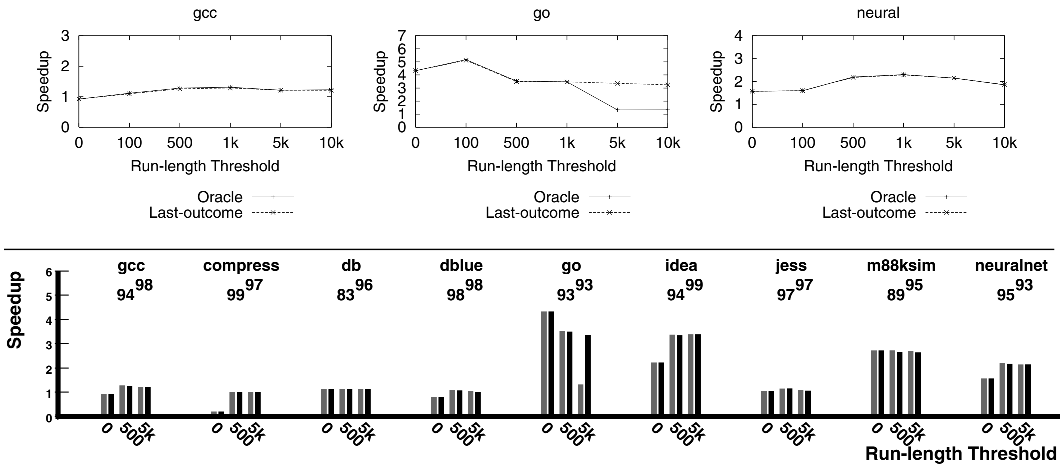

Figure 5. Speedup of the last-outcome run-length predictor (black bars) compared to the ideal predictor (grey bars), with 200-cycle overheads. Prediction accuracy (in %) is printed on top of the bars.

the module run-length could be done by recording the cycle count at the module call, and comparing it with the count after completed execution. Care has to be taken, however, to exclude overhead and time when the module is not running, e.g. swapped out in favor of a higher-priority thread. Reading the performance counters will not impose much overhead. For instance, in the AMD Athlon processors, a single instruction will read a counter register and place the result in a general purpose register [1]. A few additional instructions would be needed to store and compare instruction counts.

Since there might be a significant amount of time between the prediction and corresponding update, it is not certain that a lookup will return the result of the last invocation of the module; rather, it will be the latest that has finished. In addition, updates might not come in sequential order. However, as we will see in the next section, the accuracy of this simple predictor is very good for the thresholds of interest. In summary, we note that the design space of implementation of such predictors is large, but it is outside the scope of this paper to study their tradeoffs.

5 Experimental Results

Figure 5 shows a comparison between the speedup using oracle-determined run-lengths according to Section 3, and the last-outcome predictor described in Section 4. In the full graphs, speedup using oracle run-lengths are shown as solid lines, and speedup for the predicted lengths are shown as dotted lines. In the bar chart, grey bars are for oracle results, and black bars prediction results. Prediction accuracy is printed above the bars. Only results for overheads of 200 cycles are shown; results for 100 and 500 cycles are similar in behavior, but the differences smaller and larger in magnitude, respectively.

Overall, we can see that the predictor manages to obtain virtually the same speedup as the oracle prediction scheme, with a prediction accuracy typically above 90%. For Go, the last-outcome predictor is much better than the oracle-determined length for a threshold of 5000+. This is because the oracle at the same time disables more modules than the predictor (decreasing parallel coverage), and suffers from an increased number of misspeculations. In this particular case, the imperfection of the last-outcome predictor was beneficial! However, it occurs for a threshold higher than the best. If the predictor happens to fail on threads which are above the best threshold but below the chosen one, it is reasonable that the speedup is better for the predictor than the oracle.

Table 2. Speedup improvement.

App. *name*	*Best* *threshold*	*Improvement* *oh=100*	*Improvement* *oh=200*
gcc	1k	3%	39%
compress	500+	117%	380%
db	-	0%	0%
dblue	500	8%	34%
go	100	14%	18%
idea	500+	23%	50%
jess	100	4%	7%
m88ksim	100	1.0%	1.6%
neural	1k	17%	46%

Table 2 lists the best found thresholds for all applications at 100 and 200 cycle overheads. The improvement in speedup with the best found threshold is compared to running the programs without module run-length prediction. Note that the improvement for Compress is moot for reasons discussed earlier. The two applications marked '500+' showed a similar speedup for thresholds above 500 and up to 10000, which is the highest threshold we have used.

In summary, the speedup results at best threshold using the run-length predictor is typically within two percent of the results of an oracle. In addition, with overheads of 200 cycles, six of the nine benchmarks show a speedup improvement, which is between 7-50% compared to running all modules speculatively.

6 Systems with Limited Live Threads

We have seen that module run-length prediction is useful for preventing the creation of speculative threads that will not contribute to speedup. In this section, we show that the same technique can be beneficial for speculation systems where the number of *live threads* is limited.

As mentioned in Section 2.1, an STLP machine must store speculative values from all threads that have not yet committed. The need to handle speculative state is perhaps the main reason why proposed STLP processors allow only a low number of threads in the system. Implementing efficient speculation mechanisms with a larger number of threads than processors is a tricky problem. In addition to the storage problem, threads that are not running must take part in dependence checking, value forwarding, and might need to roll back. The non-committed threads, which we refer to as *live threads*, that exist must be visible to the speculation system even when they are not running on a processor. There must be support for at least one live thread per processor, where the running thread stores its speculative values.

There are two reasons why allowing more live threads than processors are important. The first reason is that we might want to preempt a running thread in order to run a new less speculative thread. The other, and even more important reason, is that module-level threads are of highly varying length, and therefore load-imbalance is a big problem. If we allow only one live thread per processor, any thread that finishes will tie up a processor until it becomes the head thread and can commit.

Garzaran et al. [4] present a taxonomy of methods for buffering speculative memory state, and analyzes the benefits and tradeoffs for different proposed methods. Many proposed single-chip machines, can only handle a single speculative version per processor [5, 10, 13, 16]. Hydra [6] stores state in dedicated buffers and could be extended to support more threads than processors, but the paper shows only one buffer per processor. One design from Steffan and Mowry [14] makes it possible for each processor to handle multiple threads with a specific hardware structure (speculative context) for each thread. However, none of the existing methods will scale to handle a large number of live threads due to the need for substantial additional hardware structures for each thread.

We will assume that thread preemption is possible and run simulations with support for both infinite and a limited maximum number of live threads in order to see the impact of this parameter.

As before, when a call instruction is encountered and a free speculative context is available, a new thread will be spawned. A running thread will be preempted if the new is less speculative, since completing the least speculative threads as soon as possible will allow them to commit and free up space for new threads. When the live thread limit has been reached, and a new call is encountered, our policy is to start the new thread and squash the currently most speculative one.

Simulations were run without thread-management overheads and with oracle run-length prediction, in order to isolate the effect of limiting live threads.

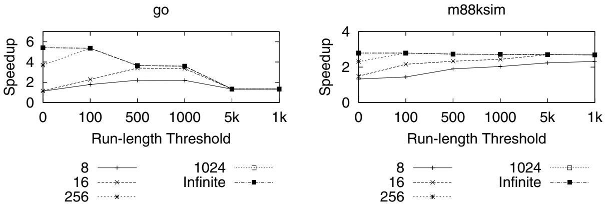

Figure 6. Benefit of run-length thresholds with limited live threads.

For some of the programs, when the maximum number of live threads is low, speedup suffers significantly – we cannot keep all finished and preempted threads in the system until they can commit. The programs that benefit are Go and M88ksim (shown in Figure 6); the others are not affected much, since misspeculations is the major problem. As the module run-length threshold is increased, fewer threads are created, and not as many speculative memory states need to be kept in the system. Consequently, the problem with limited live thread support is less significant. With better value prediction or in programs with fewer misspeculations, this technique could be more important; in simulations with perfect value prediction, we have seen that seven of the applications benefit from run-length thresholds when the number of live threads are limited.

The best threshold may be different from what is reported in the previous section. For example, Go with a maximum of 8 or 16 threads performs best at a threshold of 500-1000, compared to the best threshold of 100 found in Section 5. The combined effect of live thread limit and overheads should be considered when choosing threshold for such a system.

7 Conclusions

We have presented a new technique for reducing the impact of thread-management overhead in speculative module-level parallelism. We use the *module run-length* to determine if a new thread is to be created for the call continuation. If the run-length exceeds a certain *run-length threshold*, we create the new thread; otherwise we run the code sequentially. Empirically, we have found that 500 cycles is a good threshold for overheads in the range 100-200 cycles.

Module run-lengths are not known until the module has completed execution, but the decision to speculate must be made when the module is called. We have solved this with a module run-length predictor, which stores whether the run-length was above or below the threshold. The most recent result is used as a prediction for the next invocation of the same module.

The last-outcome predictor is shown to have a very good accuracy, between 83% and 99% compared to an oracle. In addition, six of the nine benchmarks show a speedup improvement when using run-length prediction. For overheads of 200 cycles, the improvements range from 7% to 50% compared to running all modules speculatively.

Acknowledgments

This research has been supported by the Swedish Foundation for Strategic Research under the PAMP program. The authors would like to thank Peter Rundberg and Jim Nilsson of Chalmers University of Technology, as well as the anonymous reviewers, for comments on earlier drafts of this paper which greatly enhanced the final version.

References

[1] AMD Inc. *AMD Athlon Processor x86 Code Optimization Guide*, pages 235–242. AMD Inc., 2002.

[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] 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.

[4] M. Garzaran, M. Prvulovic, J. Llaberia, V. Vinals, L. Rauchwerger, and J. Torrellas. Tradeoffs in buffering memory state for thread-level speculation in multiprocessors. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA '03)*. IEEE Computer Society, Feb. 2003.

[5] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative versioning cache. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA '98)*, pages 195–206. IEEE Computer Society, Feb. 1998.

[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] S. Hu, R. Bhargava, and L. K. John. The role of return value prediction in exploiting speculative method-level parallelism. Technical Report TR-020822-02, University of Texas at Austin, Aug. 2002.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] J. G. Steffan, C. B. Colohan, and T. C. Mowry. Architectural support for thread-level data speculation. Technical Report CMU-CS-97-188, Carnegie Mellon University, Nov. 1997.

[15] 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.

[16] 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.

[17] 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.