THESIS FOR THE DEGREE OF LICENTIATE OF ENGINEERING

Module-Level Speculative Execution Techniques on Chip Multiprocessors

Fredrik Warg

Department of Computer Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2003

Module-Level Speculative Execution Techniques on Chip Multiprocessors

Fredrik Warg

Technical Report 21L
ISSN 1651-4963
School of Computer Science and Engineering

Department of Computer Engineering
Chalmers University of Technology
SE–412 96 Göteborg, Sweden
Phone: +46 (0)31–772 1000

Contact information:
Fredrik Warg
Department of Computer Engineering
Chalmers University of Technology
Rännvägen 6B
SE–412 96 Göteborg, Sweden

Phone: +46 (0)31–772 1668
Fax: +46 (0)31–772 3663
Email: warg@ce.chalmers.se
URI: http://www.ce.chalmers.se/staff/warg

Printed in Sweden
Chalmers Reproservice
Göteborg, Sweden 2003

Module-Level Speculative Execution Techniques on Chip Multiprocessors

Fredrik Warg
Department of Computer Engineering, Chalmers University of Technology

Thesis for the degree of Licentiate of Engineering, a Swedish degree between M.Sc. and Ph.D.

Abstract

Thread-level parallelism is an increasingly popular target for improving computer system performance; architectures such as chip multiprocessors and multithreaded cores are designed to take advantage of parallel threads within a single chip. The performance of existing single-threaded programs can be improved with automatic parallelization and thread-level speculation; however, overheads associated with speculation can be a major hurdle towards achieving significant performance gains.

This thesis investigates module-level parallelism, or spawning a speculative thread running the module continuation in parallel with the called module. The results of simulations with an ideal speculative chip multiprocessor show that execution time can potentially be, on average, halved with module-level parallelism, but that misspeculations are common and that speculation overheads dominate the execution time if new threads are started for every module continuation.

In order to deal with the overhead, techniques for selectively starting new threads only when they are expected to yield useful parallelism are introduced. The first technique is aimed at removing small threads. It employs a lower threshold on module run-length; a new thread is spawned only if the run-length of the called module exceeds the threshold. In addition, run-length prediction is used to predict if the run-length will exceed the threshold. The chosen predictor typically achieves over 90% accuracy, and with this technique speculation overheads can be tolerated.

The second technique is misspeculation prediction and is aimed at bringing down total overhead. The selection of module calls used for speculation is based on whether or not spawning a new thread is expected to result in a misspeculation. When spawning threads for all continuations the total overhead is on average 336% extra cycles compared to sequential execution; with misspeculation prediction, the average overhead can be brought down to 54%.

The proposed techniques can be applied in run-time to speed up existing applications on chip multiprocessors with thread-level speculation support.

Keywords: Chip multiprocessors, thread-level speculation, module-level parallelism, module run-length prediction, misspeculation prediction, value prediction.

List of Appended Papers

This thesis is a summary of the following three papers. References to the papers will be made using the Roman numbers associated with the papers.

I. Fredrik Warg and Per Stenström. Limits on Speculative Module-level Parallelism in Imperative and Object-oriented Programs on CMP Platforms. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT 2001)*, pages 221-230, September 2001.

II. Fredrik Warg and Per Stenström. Improving Speculative Thread-Level Parallelism Through Module Run-Length Prediction. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2003)*, page 12 (abstract), article available on accompanying CD, April 2003.

III. Fredrik Warg and Per Stenström. *Reducing Misspeculation Overhead for Module-Level Speculative Execution*. Technical Report 03-07, Department of Computer Engineering, Chalmers University of Technology, April 2003. Submitted for publication.

1 Introduction

Utilizing as much as possible of the available parallelism in an application is key for a high performance computer system. To that end, modern superscalar processors exploit fine-grained instruction-level parallelism (ILP), and multiprocessor systems additionally take advantage of the parallelism in multithreaded programs.

Future performance gains from instruction-level parallelism may be costly; design complexity, diminishing returns, and technological constraints [1] make large wide-issue designs less attractive. The *chip multiprocessor* (CMP) is an emerging architectural style where several processor cores are integrated onto the same die, instead of using the growing transistor count for a single increasingly complex core. Compared to traditional multiprocessors, this new generation sports higher bandwidth and lower latency communication, thus expanding the scope of applications that can benefit from parallelization. From a programming standpoint, however, writing parallel programs is more challenging. The majority of existing applications are single-threaded, and therefore unable to use the full processing power of a CMP.

A *thread-level speculation system* makes it possible to aggressively parallelize codes when a parallelizing compiler or programmer can not determine if there will be data dependences between the threads or not. Threads are optimistically run in parallel while the speculation mechanism checks for dependences and rolls back erroneous execution if a violation is detected. Due to its fast communication, the chip multiprocessor is a good architecture for running small speculative threads; typical thread sizes are a few hundred up to a few hundred thousand instructions. Several proposed CMPs integrate support for thread-level speculation [4, 6, 7, 9, 12, 14, 16, 17].

A variety of methods for creating speculative threads have been investigated. In this thesis I explore *module-level parallelism* (MLP), or creating a new thread for the module (i.e. function, procedure or method) continuation when execution reaches a call instruction. An appealing advantage of MLP is that call and return instructions are easy to identify, making it possible to create threads in run-time, and eliminating the need for recompilation. The potential of module-level speculative execution has been studied in [2, 11, 13]; however, **I** was the first to compare C and Java programs.

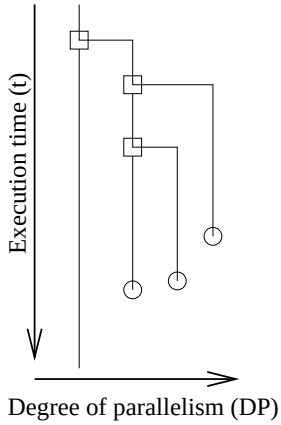
Thread-management overhead and frequent misspeculations are known to be major obstacles for thread-level speculation. **I** identifies a mismatch between the module granularity and the thread-management overhead of CMP proposals. The impact of overheads and prevalence of misspeculations are affected by how the program is split into threads. **II** contributes with a technique which will not start a new thread for a continuation when it is not expected to yield any meaningful parallelism, and will reduce the number of small threads. **III** describes a technique used to predict which module invocations will cause misspeculations, and greatly reduces the total overhead. Another method that can reduce the number of misspeculations is value prediction, or guessing the value produced by a load or function return; predictions are usually based on execution history. In **I**, the limits on speedup with value prediction is investigated.

The methodology used throughout the thesis consists of two stages of simulation.

First, traces from sequential execution are captured using Simics [8], a full-system instruction-set simulator. The traces are used in my trace-driven simulator for module-level speculative execution, which implements all techniques discussed in the thesis and collects relevant statistics.

Section 2 of this thesis discusses the potential of, and problems with, module-level speculation; it summarizes the findings in **I**. Then, Section 3 presents the techniques for mitigating the impact of speculation overhead proposed in **II** and **III**. Finally, Section 4 concludes with discussion and some ideas for future work.

2 Limits on Speculative Module-Level Parallelism



The figure shows an example of module-level parallelism: boxes mark module invocations (possible thread spawn points), circles show module returns (thread completion points), and the length of the vertical lines represents the relative execution time of the modules. The degree of parallelism at any point in the execution, DP(t), equals the number of simultaneously running threads, which is affected by the amount and length of modules, as well as when they are called. In the figure, DP(t) varies from one to four as new threads start or are completed.

The speculation system handles creation of new threads, which includes transferring initial state (register values, PC) from the existing thread. All results produced by speculative threads are buffered while the thread is executing; meanwhile dependence checking is handled by hardware added to the memory system, for instance as a part of the coherence protocol [5]. If a flow dependence (a read dependent on a previous write) violation is detected between two threads the one which has read an erroneous value must roll back, i.e. all buffered results are thrown away (or squashed) and execution is restarted. If the thread completes successfully, it can commit – the buffered results are merged with main memory.

Even if the average DP(t) is good, the amount that can be successfully exploited may be limited due to:

- Dependence violations caused by shared variables or return values.
- Limited number of processors preventing full use of the available parallelism.
- Thread-management overheads imposed by the speculation system when starting new threads, rolling back execution, and committing finished threads.

In order to determine whether module-level parallelism is a good candidate for thread-level speculation, **I** sets out to establish the average DP(t) in a typical application, and to study the impact of these potentially limiting factors.

Nine applications written in C and Java are studied. Since the aim is to expand the scope of programs that can make use of thread-level parallelism, the chosen applications are integer programs that have not been successfully parallelized by compilers. **I**

shows that the module-level parallelism is typically between two and six, with a geometric mean above three for both C and Java applications. When taking dependences into consideration, the mean speedup goes down to around two. Practically all of the remaining parallelism can be exploited with an eight-way chip multiprocessor.

A stride value prediction scheme is evaluated as a means to decrease the number of violations for return values. When a new thread is started, the return value for the module is predicted to become the same as the result returned from the previous invocation of the same module, plus the difference between the previous two. This simple prediction scheme is found to correctly predict 20%-80% of the return values. Since many of the remaining returns are hard to predict, for instance random values, the performance of this scheme is respectable. However, since misspeculations through memory accesses dominate, **I** suggests more research in this area. The intuition that Java programs would show fewer misspeculations through shared variables and more parallelism due to the programming style proved unfounded, no significant difference between the C (imperative) and Java (object-oriented) applications was found.

Speculative module-level parallelism has also been investigated in [2, 11, 13], but **I** is the first to directly compare C and Java programs as well as quantify the limits and thus the improvement potential of value prediction in this context.

In summary, the results in **I** show that there is module-level parallelism to exploit with small-scale chip multiprocessors, but that dependences and thread-management overhead are major obstacles; many threads are too small to amortize the speculation overheads of proposed speculative chip multiprocessors. However, as will be discussed in the next section, careful selection of calls used for spawning new threads can improve upon the situation.

3 Improving Module-Level Speculative Execution

It was established in **I** that creating new threads for all module invocations will result in bad performance when overheads are taken into account, since a large number of modules are small compared to overheads in current speculation system proposals. Also, the number of misspeculations increase as we add more processors, so simply throwing in more processing resources will not solve the problem. Other measures are needed, and the following sections present techniques which attack the problem by being selective when deciding when to spawn new threads; not all module invocations are suitable as thread spawn points. The common trait of the techniques is that they can be implemented in run-time and thus do not require recompilation.

3.1 Module Run-length Prediction

As mentioned, a key problem for module-level speculation is to create threads of appropriate size. With many short threads, the execution time can be dominated by the overheads imposed by the speculation system for starting new threads, rolling back execution after a violation, committing completed threads and switching between threads. Long threads, on the other hand, are more likely to be squashed and it can be difficult to

find enough threads to utilize the available processors and gain a good speedup. Hammond et al. [6] found that threads of size 300-3000 instructions are suitable if overhead is in the range of 10-100 cycles. It is not only the length of threads that is important, but also how much overlap there is between a thread and the child it spawns. If the called module is small the overlap, and thereby the gain in parallelism, will also be small.

Compiler inlining can be used to avoid speculation for small modules; however, there are two reasons for investigating alternative methods. First, the compiler cannot always compute the number of dynamic instructions in a module due to input-dependent control. Second, relying on the compiler will ruin the MLP advantage of being suitable for run-time parallelization.

In **II**, a threshold on the module run-length is used to decide when a thread should be spawned. Imposing a lower limit on the run-length for creating new threads will avoid speculation in situations where there would be little parallelism to gain from starting a new thread, and will reduce the number of small threads observed in **I**. Module run-length is the measured execution time of a module without speculation overheads. Execution time for a module called within the measured module is included if no speculative thread was created for the continuation, and otherwise excluded. If the run-length is found to be below a certain threshold, the module invocation will not result in a new thread for the continuation.

I found that most of the impact on speedup of overheads in the range of 100-500 cycles can be eliminated by using a run-length threshold, assuming a priori knowledge of the run-lengths. The best threshold is around 500 cycles for most applications when thread-management overheads are 200 cycles. Without the technique, three of the benchmarks show a slowdown with 200-cycle overheads, these slowdowns are eliminated.

However, since the run-length is needed before the module has been executed, perfect knowledge about the run-length at the time of the speculation decision is not possible. Instead, I propose to use prediction of the run-length based on measurements during earlier invocations of the same module. In **II** it is shown that a simple last-outcome predictor can achieve an accuracy of 83% to 99% with virtually the same speedup improvements as with perfect a priori knowledge of the run-lengths. The last-outcome predictor will guess that the run-length of the called module will be on the same side of the threshold as it was for the previous invocation.

Six of the nine benchmarks show speedup improvements of between 7% and 50%, two are unaffected by 200-cycle overheads, and for the last application a large slowdown is eliminated. With run-length prediction, thread-management overheads can be tolerated when exploiting module-level parallelism.

3.2 Reducing Misspeculation Overhead

Even if the speedup looks good the total overhead in the form of thread-management overhead, or extra time imposed by the speculation system, and execution overhead, time spent reexecuting squashed code, can be a dominant part of the execution time. In

III, I show that the total overhead when starting new threads for all module invocations on an eight-way machine with 100-cycle thread-management overheads is, on average, as high as 336% of the serial execution time for the same application. If energy efficiency or resource utilization in a multiprogrammed system is considered, it is clear that the speedups achieved often come with a high price in wasted resources.

The amount of roll-back and execution overhead is determined by the amount of misspeculations, but also the precision of the roll-back; more squashing means more reexecution and more restarted threads. Most speculation system proposals implement roll-backs by squashing the entire violating thread and all threads that depend on it – a policy which is relatively easy to implement. However, a checkpointing mechanism such as the one described by Olukotun et al. [11] makes it possible to save checkpoints and return to an earlier point in the violating thread instead of squashing the whole thread. In **III** it is shown that checkpointing could potentially be useful in module-level speculative execution, five of the benchmarks have notably worse speedup without checkpointing.

III also evaluates two techniques for reducing the number of misspeculations by selectively excluding module invocations from spawning new threads. In contrast to schemes attempting to synchronize dependent instructions or threads in order to remove flow dependence violations [3, 10, 15] this technique removes misspeculating threads entirely, thus getting rid of all associated overhead. The better of the techniques – called misspeculation prediction – manages to bring down total overhead from a mean 336% to a mean 54%, with improved speedup for about half of the applications but lower speedup for three. Misspeculation prediction works as follows: when the speculation system detects a violation, the module call from which both threads involved in the misspeculation originates (their common ancestor) is marked. When the same module is called again, no new thread is created for the continuation, hopefully delaying the early load enough to avoid a repetition of the misspeculation. A number of prediction policies are evaluated in **III**; a simple last-outcome predictor for each module is found to perform best in addition to being relatively easy to implement.

Since some applications with few misspeculations suffer worse speedups when misspeculation prediction is enabled, **III** also suggests using misspeculation prediction only when the number of misspeculations is high. The prevalence of misspeculations is measured as the ratio of the number of squashed threads to the number of started threads. If misspeculation prediction is enabled only when this ratio is above 0.6, the speedup is better or the same for all applications as when creating new threads for all continuations, but the overhead is down to an average 89%.

In summary, it is possible to remove most of the overhead while maintaining or even improving speedup, with a relatively simple and low overhead prediction technique that can be implemented in the run-time system as part of an existing speculation system.

4 Discussion and Future Work

The focus of this thesis is module-level speculative execution in the context of chip multiprocessors with hardware support for thread-level speculation. It brings together the increasing interest in exploiting thread-level parallelism with the need to maintain high performance for existing single-threaded applications and codes hard to parallelize with compilers or by hand. The run-time techniques make it possible for sequential programs to take advantage of the extra resources in a chip multiprocessor. In **I** I looked at the potential of module-level parallelism as well as the problems involved in translating this potential to performance improvements. In **II** and **III**, I contribute with techniques making module-level parallelism more tolerant to overheads.

An area that has not been investigated thoroughly is the trade-off between ILP and speculative thread-level parallelism. Dividing a program into relatively small speculative threads may eat into the ILP extracted by superscalar cores; modern cores can have instruction windows of more than a hundred instructions. The trade-off is especially interesting to study as a range of CMP architectures with different number and complexity of the integrated cores have been presented and are starting to emerge on the market.

Communication and cache behavior is another issue I have not yet looked into. Speculative execution puts additional pressure on the memory system, but on the other hand, recent work has shown that even misspeculating threads can be useful for their prefetching effect. How this plays out in the context of MLP is another interesting topic.

Finally, proposed speculation systems are quite inflexible, speculative threads can typically not be preempted and there are limitations on the number of live threads (i.e. number of threads that can be simultaneously handled by the speculation system). Hopefully, there are still improvements in the way thread-level speculation is implemented waiting to be discovered.

Acknowledgments

I would like to thank my advisor Per Stenström for support and encouragement, for ideas and discussions, and for teaching me what good research is all about. I also want to thank my family and friends for all their support, and the people at the department of Computer Engineering who make it an enjoyable and interesting place to work. I especially want to mention some past and present Ph.D. students in the high-performance computer architecture group (in alphabetical order): Björn Andersson, Ulf Assarsson, Cecilia Ekelin, Magnus Ekman, Jochen Hollman, Jonas Jalminger, Martin Kämpe, Thomas Lundqvist, Jim Nilsson, and Peter Rundberg.

This research has been supported by a grant from the Swedish Foundation for Strategic Research (SSF) under the ARTES/PAMP program. Equipment grants from Sun Microsystems have been instrumental to support the many computer simulations needed to carry out this work.

References

[1] V. Agarwal, M.S. Hrishikesh, S.W. Keckler, and D. Burger. Clock rate versus ipc: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*, pages 248–259. 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, October 1998.

[3] 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, February 2002.

[4] 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, October 1999.

[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, February 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, October 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] 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)*. IEEE, May 1997.

[11] K. Olukotun, L. Hammond, and M. Willey. Improving the performance of speculatively parallel applications on the hydra CMP. In *Proceedings of the 1999 International Conference on Supercomputing (ICS '99)*, pages 21–30. ACM Press, June 1999.

[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. 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, October 1999.

[14] 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.

[15] 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)*. IEEE Computer Society, February 2002.

[16] 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, February 1998.

[17] 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, October 1996.

Paper I

Limits on Speculative Module-level Parallelism in Imperative and Object-oriented Programs on CMP Platforms

Reprinted from

Proceedings of the International Conference on
Parallel Architectures and Compilation Techniques (PACT 2001),
September 2001.

Limits on Speculative Module-level Parallelism in Imperative and Object-oriented Programs on CMP Platforms

Fredrik Warg and Per Stenström
Department of Computer Engineering
Chalmers University of Technology
SE–412 96 Göteborg, Sweden
{*warg, pers*}*@ce.chalmers.se*

Abstract

This paper considers program modules, e.g. procedures, functions, and methods as the basic method to exploit speculative parallelism in existing codes. We analyze how much inherent and exploitable parallelism exist in a set of C and Java programs on a set of chip-multiprocessor architecture models, and identify what inherent program features, as well as architectural deficiencies, that limit the speedup. Our data complement previous limit studies by indicating that the programming style – object-oriented versus imperative – does not seem to have any noticeable impact on the achievable speedup. Further, we show that as few as eight processors are enough to exploit all of the inherent parallelism. However, memory-level data dependence resolution and thread management mechanisms of recent CMP proposals may impose overheads that severely limit the speedup obtained.

1. Introduction

While instruction-level parallelism has been a good source to boost performance of processors over a long period of time, this source is now getting exhausted. The major reasons are the difficulties in supporting huge instruction windows as well as in speculating beyond control-flow dependences. As a remedy, several recent papers have proposed support for speculative thread-level parallelism (STLP) in the context of chip-multiprocessors (CMPs) [6, 16]. A chip-multiprocessor with support for thread-level data speculation allows programs partitioned into threads to correctly execute in parallel even if it is not ascertained that the threads are indeed data independent. If it turns out that a data dependence violation occurs, the speculation support will detect it which permits the speculation system to abort and re-execute the threads in a way that respects sequential semantics.

The most popular form of STLP to exploit has been loop-level parallelism. While impressive parallelism can be obtained in numeric applications with loops that contain few loop-carried dependences, the poor parallelism coverage or lack of do-all loops in general integer applications severely limit this approach [12]. On the other hand, module-level parallelism, i.e., parallelism across function, procedure, or method invocations, is potentially a more general and useful form of STLP. First, it is very simple to identify the thread boundaries; new threads are created at module invocations, and terminated when they reach a return. Second, we avoid the control dependence problem we encounter in, for instance, loop-level speculation. Presumably most importantly, however, is that modules are used frequently as the key abstraction mechanism in object-oriented programs in particular but also in imperative programming styles.

The first goal of this paper is to understand to what extent the programming style – imperative versus object-oriented – affects the *inherent* speculative module-level parallelism. We do this by considering a set of C and Java programs and carry out a speedup limit study assuming an idealized machine model. This model has an infinite number of processors, it supports perfect prediction on return as well as memory values, and it imposes no overhead on thread management or inter-thread communication. While Oplinger *et al.* [12] present a limit study based on a similar idealized machine model, they didn't consider Java programs.

The most important result we gained from the experiments on the idealized model is that there is a fair amount of module-level parallelism in C and Java programs. The question is how to best exploit it in terms of appropriate architectural support. We separate out a number of concerns through a series of successively refined architectural models as follows. The first issue we study is to what extent data dependences between threads (on return or memory values) limit the speedup obtained. More effective encapsulation of data in objects would speak in favor of an object-oriented

style of programming. It is interesting to see whether this indeed will result in fewer data dependences and higher speedup limits when comparing C and Java programs.

Another motivation is to see whether simple value prediction schemes suffice or research into more sophisticated value prediction schemes are warranted. A third issue that we address is how much machine resources are needed to exploit the inherent parallelism. We address this issue by studying the speedup limit as a function of the number of processors and again whether the expected heavier use of modules in object-oriented programs would lead to more scalability. Finally, we also address to some extent how thread management overheads impact on the achievable speedup to see whether research into more effective support is warranted and what this support should target. One interesting aspect is how well the granularity of parallelism in terms of common module sizes matches the overheads incurred in recent CMP proposals. While [12, 11, 2] have also studied the potential of module-level parallelism in C and Java programs on CMP platforms, none of them has explicitly compared the nature of the module-level parallelism inherent in C and Java programs.

The main contribution of this paper is the insights into the inherent and architectural limits on the speedup for imperative versus object-oriented programs in a single consistent framework. Our most important findings are:

- Overall, we didn't notice any significant qualitative differences between C and Java programs suggesting that the programming style has a minor effect on the amount of parallelism to be exploited.
- The inherent module-level parallelism in applications is typically not more than four to eight suggesting that small-scale CMP or multi-threaded cores are enough to exploit all of the available parallelism.
- Most of the codes do not benefit from more advanced return value-prediction schemes than stride and last-value prediction suggesting that current predictors fare pretty well.
- The granularity of modules typically don't match the overheads in recent CMP proposals. In addition, the accuracy of memory value prediction schemes is a major inhibitor to decent speedups. This suggests that more research into better machine models and memory-value prediction schemes are warranted.

In the next section, we introduce the execution model and the series of architectural models used to identify the limits on module-level parallelism. Then in Section 3, the experimental setup along with the benchmark applications used are introduced. The experimental results are provided in Section 4. We put the work in perspective of related work in Section 5 along with an outlook before we conclude in Section 6.

2. Execution and architectural models

In this section, we first introduce the execution model as seen by the software and then introduce the machine models used to identify what the limits on achievable speedup assuming the execution model are.

2.1. Module-level execution model

The true advantage of module-level parallelism lies in its simplicity. In its simplest form, a new thread is spawned at each module (i.e., function, procedure, or method). To respect sequential semantics on a module invocation, the old thread of control executes the module, whereas a new thread is spawned that speculatively executes the code after the module call as shown in Figure 1. If another module call is encountered, a new speculative thread that executes the continuation of the module will once again be created. In order to respect sequential semantics, the new thread will be more speculative than the thread from which it was created (i.e. it would execute after the original thread in sequential execution), but retain the same relationship as its parent with respect to all other speculative threads. All threads must commit in sequential order; in other words, a thread cannot commit until all less speculative (earlier in sequential execution order) threads have already committed.

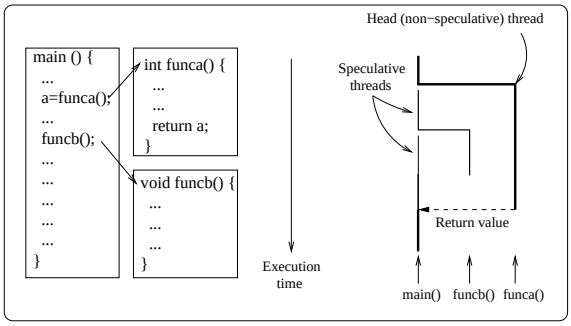


Figure 1. Execution model

2.2. Architectural models

The four models introduced gradually encounter more of the architectural limitations associated with recent CMP proposals. For each model, we first specify what limitations it encounters and then discuss what issues will be addressed with it.

Model 1: Inherent module-level parallelism

A speculative thread will successfully terminate as long as no data dependences are violated with threads that would

precede it according to sequential semantics. Then the upper bound on the speedup is dictated by the control dependences between subsequent module invocations. We wish to understand the scalability of module-level parallelism in terms of how severely the control dependences set in. The first model therefore assumes a machine with an infinite number of processors and that no data dependences will be violated and cause rollbacks. In addition, thread management and inter-thread communication costs are zero.

This model provides important insights into the difference of imperative and object-oriented programming styles. One hypothesis is that an object-oriented style of programming would lead to more scalability in exploiting module-level parallelism. This is one of the hypotheses we will test using this model.

Model 2: Impact of data dependences

With this model, we are interested in how data dependences between threads limit the achievable speedup and whether proposed support in terms of forwarding and value prediction in the recent literature is enough.

There are two classes of data dependences: flow and name dependences. In this as well as in the subsequent architecture models, we assume that name (anti- and output) dependences can be resolved through renaming. In CMPs with speculation support this is done by keeping speculative state in the cache until the thread can commit, which involves flushing the speculative state back to the memory. As Steffan and Mowry have found [16], for the small-grain threads usually considered for CMPs, the available cache space seems sufficient to host the speculative state created.

On the other hand, flow dependences may have a severe impact on the achievable speedup through module-level speculation since a data dependence violation will result in a rollback. If a thread has computed a value before a more speculative thread reads it, the most recent value will be forwarded to the more speculative thread; but, if the value is computed after the more speculative thread performs the read, a flow dependence violation occurs. After a violation, the more speculative (violating) thread will rollback execution in order to maintain correct sequential execution.

The model we assume is capable of perfect rollbacks, which means that the thread causing the violation will be able to restart execution exactly at the load instruction causing the violation. However, threads started by the violating thread after the erroneous instruction are squashed.

Flow dependences take two forms: flow dependences through memory and return values. To separate out the relative frequency of each category, we experiment with six alternatives:

- Heap accesses have either (1) perfect value prediction or (2) none at all. Perfect value prediction means that the prediction is always correct, and consequently we never get any memory-bound dependence violations.

- Value prediction for return values comes in three flavors: (1) Perfect return value prediction (RVP) is once again always correct; (2) stride RVP is supported by a table storing a last value and a stride value for each procedure; the table is of unbounded size. RVP buffers are updated in execution order, which is not necessarily in the same order as in the sequential execution. Additionally, it might happen that a finished thread updates the value predictor and then gets squashed, resulting in predictor pollution; one could say that the value predictor is speculatively updated. (3) The third option is no return value prediction.

With this model, we are able to answer questions related to the relative importance of memory versus return value flow dependence violations and how they relate to the programming style. One hypothesis would be that object-oriented programs tend to better encapsulate memory-bound flow dependences whereas dependences caused by return values become more critical. In addition, it is possible to pinpoint whether it would make sense to focus future research on more sophisticated value prediction schemes for STLP.

Model 3: Impact of limited processing resources

While the scale of CMPs will increase with increased integration, it may not make sense to charge too many resources to thread-level parallelism in trading off number of processors versus issue-width for example.

In the third model, we study to what extent the number of processors limit the speedup. When the number of available threads exceed the number of processors, priority will be given to threads based on sequential execution order. New threads with higher priority will preempt more speculative threads if needed.

One problem noted in the Hydra project [6] regards the speculative state stored in the caches. To avoid having to save the cache state, it is not possible to preempt a speculative thread until all the preceding speculative threads have committed. This may severely limit the speedup obtained for module-level speculation. If preemption is impossible, a new thread cannot be created when all processors are in use, unless a more speculative thread occupying one of the processors is squashed, wasting the work it has already done. An even more serious consequence would be load-imbalance problems. If a large thread is running, completed more speculative threads can neither commit, nor yield the processor, and therefore the processor will remain idle until the large thread finishes. We do not, however, impose this limitation as our aim is to establish an upper-bound on the

available parallelism. While one would have to address this issue, it does not appear as a hard problem.

Model 4: Impact of thread-management overhead

In the preceding models, we have assumed that threads can be spawned, committed, and rolled back in zero time. On recently proposed CMPs such as Hydra, the overheads imposed by these operations are not negligible. To what extent the overheads have a significant impact on the speedup obtained is strongly connected to two application parameters: the number of flow dependence violations and the module granularities. Our goal with this model is to factor in these overheads to identify what mechanisms would have to be further researched to come up with machine models better adapted to module-level parallelism.

3. Methodology and benchmarks

In this section, we first explain how the simulations were done, and then present the benchmarks used in our experiments.

3.1. Simulation tools

All results presented in the following sections are obtained from our trace-driven simulation tool. The simulation tool implements all architectural models described in the previous section. The tool runs a program much as it would be run on a real machine with speculation support, that is, threads are run in parallel with run-time dependence checking. If a dependence violation is detected, the violating thread is rolled back and subsequent threads squashed. It is possible to set a maximum number of active threads (number of processing elements) or to let every available thread run simultaneously.

As opposed to a real machine, only instructions of importance for the simulation are supported; i.e. module calls, loads and stores, returns (including the actual return value) and an instruction marking that the return value was used. These events are included in the traces. A virtual timer, which keeps track of the number of instructions executed between such events, is associated with each thread.

Traces with all information needed by the analysis tool were obtained by running the programs sequentially on a system-level instruction set simulator, Simics [8]. Simics makes it possible to run applications and OS in a simulated environment, and to capture memory accesses and register contents without introducing any overhead in the application. Another feature of Simics we use is to annotate the programs to make a call-out from the application to the memory system simulator to mark an event in the program in the same way as any memory system event. This feature was used to mark module calls and returns, as well as the first occurrence of return value use.

The simulated processor is a single-issue in-order SPARC v8. The memory system is assumed to be perfect, loads and stores are always available for use in the next clock cycle. This means that the simulated system always completes one instruction each cycle.

Realistic processor core and memory hierarchy models would affect the run-time of each module: ILP would decrease thread execution time on a modern superscalar core, and an imperfect memory hierarchy would increase execution time. There are many issues affected by the memory hierarchy, for instance the impact of inter-thread communication, speculative state management, context switch overhead, sharing overhead if separate caches are used, and increased bandwidth requirements because of speculation. To determine exactly how this would affect speedup, a cycle-accurate simulator of a CMP with speculation support is needed. For the time being, we have omitted these points of the design space in favor of what we consider to be more fundamental issues. Processor and memory hierarchy considerations are very important, however, and an interesting topic of future papers.

The programs were compiled with the GNU Compiler Collection (GCC) 2.95.2 with full optimizations. In a Java Virtual Machine (JVM) environment, the execution of a Java program includes class loading and verification, Just-In-Time (JIT) compilation and/or interpretation, and garbage collection. In our measurements, the Java programs are run without a JVM. Instead, they are compiled to native executables, which means neither class loading and verification, nor interpretation or JIT-compilation occurs. Furthermore, garbage collection has been disabled. Since our intention is to find the parallelism inherent in the applications, and not to evaluate the Java run-time system, this method makes sure our measurements only contain code execution. In addition, it gives a fair comparison between Java (Object-Oriented) and C (Imperative) codes, since GCC can be used to compile all of the benchmarks. It should be noted, however, that the Java compiler is still under development, and optimizations are not as good as for the C compiler, so in comparison, the Java program instruction counts might be somewhat higher than what would be achieved with a production-quality compiler.

Only modules in the actual application are marked by the compiler. This means we do not speculate on library (or class library for Java) calls. Such functions are run inside the caller thread. Another noteworthy detail is that exceptions and I/O operations would inhibit the ability to run threads speculatively in a real machine. These events are rare in our benchmarks, so they have not been considered in the simulations. Artificial dependences through the stack from the sequential execution have been removed.

Figure 2 summarizes the simulation process: First the application is compiled with GCC, and annotations to make

call-outs to Simics are inserted; then it is run on top of Simics, generating the trace; and finally the trace is 'executed' on the architectural models that collect the statistics we will present in the subsequent section.

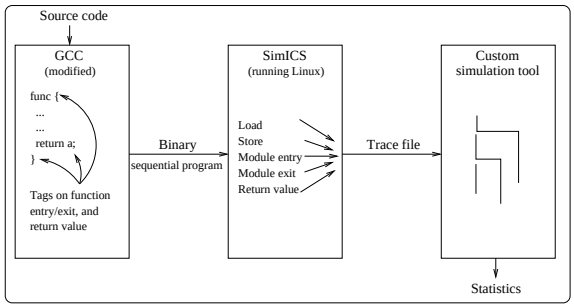


Figure 2. Our toolchain

3.2. The benchmarks

We have selected ten benchmarks, four written in C (imperative) and six written in Java (object-oriented). The C benchmarks are from the well-known SPECint95 benchmark suite and have been used in earlier STLP limit studies [12, 9]. From the eight SPECint95 benchmarks, we chose four that based on the earlier studies seem to represent typical behavior.

Three of the Java benchmarks are from SPEC JVM98. Unfortunately, the rest of the benchmarks in the suite did not include source code. Instead, we included two benchmarks from jBYTEmark (also used in [2]) and a constraint solver benchmark.

In order to keep simulation times down, we had to restrict the size of the input data sets. While the data sets are small, there are still plenty of module calls to speculate on. We have tried to make sure this restriction does not affect the behavior of the programs (for instance resulting in large initialization phases); however, it cannot be ruled out that larger input sets could affect the result on some of the benchmarks.

Table 1 briefly explains what each benchmark does. It also includes dynamic instruction and module counts, as well as average module size, for the applications. However, it should be noted that the average module size can be a bit misleading; module sizes vary greatly. In Section 4.4 we will see that that a majority of modules are less than 100 instructions in all but two of the programs.

4. Experimental results

In this section, we present the results of our experiments on the set of models described in Section 2.2 using the methodology in Section 3. We begin with studying the upper bound on the module-level parallelism in Section 4.1 followed by the impact of data dependences in Section 4.2, impact of limited processing resources in Section 4.3, and finally impact of thread-management overhead in Section 4.4.

4.1. Limits on the inherent parallelism

Figure 3 shows the speedup for our benchmark applications with perfect (i.e. always correct) value prediction both for return values and all memory loads. The harmonic mean for both groups (C and Java) of applications is also included. The speedup under ideal machine conditions is only limited by the module-level parallelism inherent in the program structure as constrained by the control dependences, i.e., how often and when modules are called. Figure 3 therefore serves as a fundamental limit for MLP, given the simplistic execution model where we begin speculation whenever a module call is encountered. The only way to find more MLP would be to speculate on module calls, i.e. speculatively call modules before execution has reached the point of the call, which introduces the element of control-speculation. To some extent, it could also be possible to use compiler transformations to rearrange the module calls in a more advantageous way, i.e. to increase the overlap of module execution.

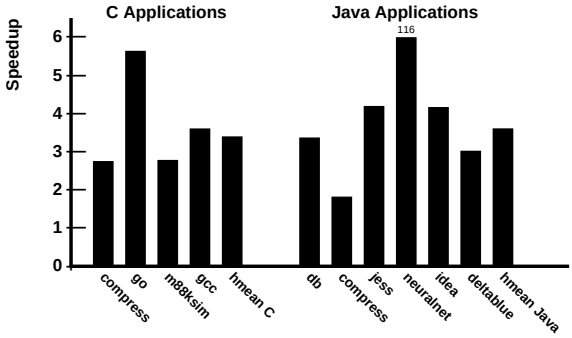


Figure 3. Speedup on the ideal machine with perfect memory and return value prediction

A noticeable fact is that the speedup without the impact of dependences is not spectacular, with a harmonic mean of 3.4 and 3.6 for the C and Java applications, respectively. This means that the module calls are not arranged in such a way that there will be a large overlap of modules even if all calls are parallelized. A contributing reason as to why we do not get a large additive effect is that the majority of modules are small. However, if some of the parallelism can be extracted with reasonable effort, it might still be a useful proposition given the simplicity by which this parallelism can be extracted from existing programs.

We can also see that there is no significant difference between the Java and C application with respect to potential MLP, the mean speedup is almost exactly the same for both

Table 1. The benchmark applications

Name	*Origin*	*Description*	*#Instructions (dynamic)*	*#Modules (dynamic)*	*Avg. instr./mod. (dynamic)*
C Applications					
compress	SPECint95	Unix compress	1.4M	21k	67
go	SPECint95	Plays the game of Go	1.4M	1.1k	1190
m88ksim	SPECint95	A chip simulator	2.2M	0.5k	4767
gcc	SPECint95	GNU C Compiler 2.5.3	13M	54.5k	237
Java Applications					
db	SPEC JVM98	Simple database	13M	4.9k	2644
compress	SPEC JVM98	Compress (Java port)	2.7M	31.5k	84
jess	SPEC JVM98	Expert system	16.3M	25.8k	633
neuralnet	jBYTEmark	Neural network	4.2M	2.6k	1626
idea	jBYTEmark	En/decryption	35.7M	12k	2966
deltablue	Sun Labs	Constraint solver	2.6M	12.5k	208

programming styles.

The reason for the high speedup (116) in NeuralNet is that a number of modules are called repeatedly inside a main loop, encompassing the entire program except for a short initialization phase. Thus, the main loop uncovers large amounts of MLP.

4.2. Impact of data dependences

The previous model predicted speedup under the assumption that value prediction on return as well as memory values is perfect. Perfect value prediction is of course not possible to attain, so the first question on our trek towards a realistic machine model is: how would value predictors with reasonable implementation complexity affect speedup?

In Figure 4, we show how memory load value prediction (MVP) affects performance. For each application the left bar, labeled (P), is the speedup with perfect MVP, while the right bar, labeled (N), indicates speedup with no MVP. The difference in height thus indicates the potential of memory load value prediction. The two rightmost bars once again show the harmonic mean.

The lack of memory value prediction has a substantial impact on some of the benchmarks. For instance, most of the massive potential in the NeuralNet benchmark disappears. NeuralNet contains numerous shared data structures that are continuously updated in each iteration of a main loop; therefore, this main loop is not possible to parallelize. The remaining parallelism comes from partial overlap of modules within a loop iteration. The key methods in NeuralNet do contain a good amount of loop-level parallelism, which cannot be exploited with the MLP-only approach. In [2], the authors have extracted module-level parallelism from this application by recoding it, converting loop-level parallelism to MLP.

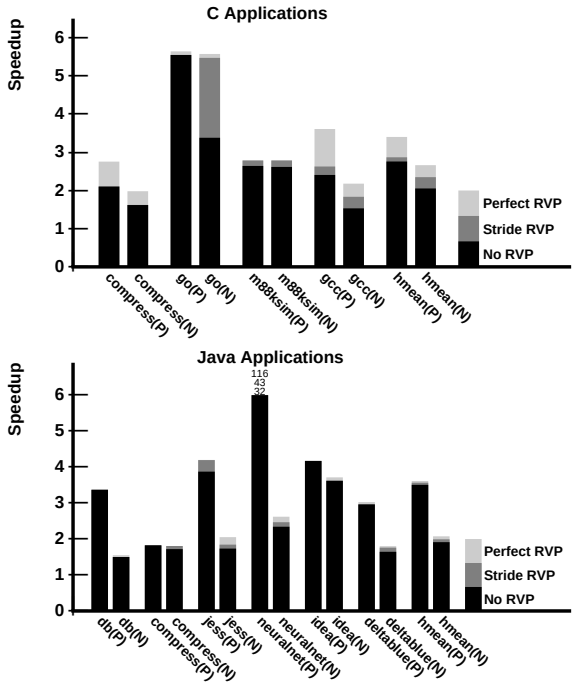


Figure 4. Value prediction: the left bar (P) for each application has perfect memory value prediction, the right bar (N) has no memory value prediction

The shaded vertical sections on each bar in Figure 4 show the impact of return value prediction (RVP). Three policies are presented: no RVP, stride RVP, and perfect RVP. For the no RVP policy, modules are still run speculatively, but a rollback always occurs to the point where the return value is used. The gap between no RVP and perfect RVP will reveal the potential benefits of return value

prediction. We also included a known and computationally simple value predictor as an indication of the predictability of return values; the stride predictor, which predicts the next value as the last value plus the difference between the two last values. Another obvious candidate would be a last-value predictor, however, they both catch the most obvious case of a function that almost always returns the same value.

Return value prediction seems to make sense in some of the benchmarks, but surprisingly, in many of the programs most of the parallelism can be exploited without RVP, since a large portion of the modules either do not produce a return value at all (void modules), or produces a return value which is never used. If one would choose a scheme without RVP, a speculation system could catch and rollback modules in the cases where the return value is indeed used, but a better way would be to have the compiler mark all calls to void modules and those whose return value is not used, since this can be determined statically.

The simple stride value predictor has been observed to perform reasonably well in most applications, successfully predicting between 20% and 80% of return values in seven of the ten applications. Some applications, notably Idea and NeuralNet, did show a very large percentage of mispredictions. It turned out to be because of heavy use of a random number function during initialization (which is a small part of the total execution time). It would be useful to be able to selectively disable speculation for such cases, where obviously no value predictor can be expected to perform well.

The execution time in Idea is concentrated to one module which handles encryption/decryption. Most of the speedup we can see for this program is because of overlap of two iterations of encryption and decryption (four calls to this module). Although it is written in Java, it has the structure of an imperative program, which is not surprising considering the fact that it is converted from C. NeuralNet and Java Compress are also originally C programs.

Our initial belief was that the object-oriented (Java) programs would exhibit more parallelism than the imperative (C) at this point for two reasons: the object-oriented programming style encourages more frequent use of module calls, and the use of data encapsulation would result in fewer memory dependences. However, our results do not indicate any such difference (Figure 4). There seems to be a small difference when it comes to return values, the speedup of the C programs are slightly more affected by rollbacks due to return value mispredictions.

In summary, two important lessons can be learned from this experiment: a simple return value predictor will suffice in most cases, and a good memory load predictor would be very useful. In the rest of this paper, we will assume no value prediction on memory loads, and stride value prediction for return values, since we feel that this represents a design choice of reasonable complexity today.

4.3. Impact of limited processing resources

In Figure 5 we can see the speedup for our applications running on a machine with limited processing resources. The model is still ideal in the sense that we assume the processing elements (PEs) on an *n*-way machine can always be utilized executing the *n* least speculative threads. A more speculative thread will be preempted, without penalty, if a new less speculative thread arrives; execution will be resumed, however, where it was preempted the next time the thread can be rescheduled on a PE.

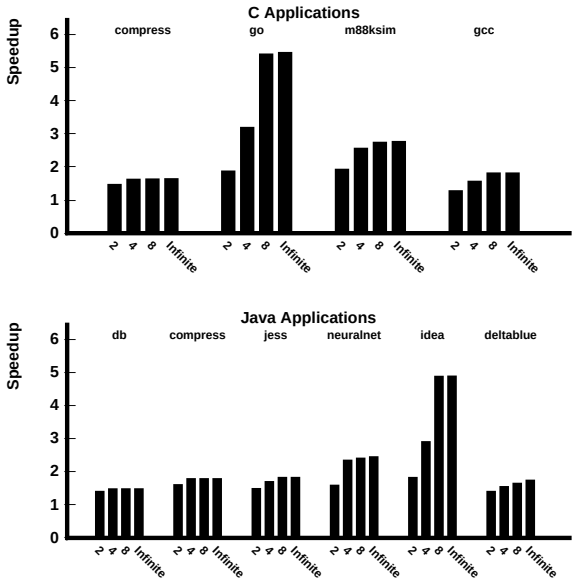


Figure 5. Speedup with 2, 4, 8 or an infinite number of PEs

With this model, we can see that virtually all potential speedup can be utilized with only eight PEs. In fact, many of the benchmarks do not benefit significantly from more than four PEs. This is good news, since it shows that parallelism is in general not concentrated to a limited part of the execution; rather, a limited number of PEs are busy working most of the time.

This data suggests that all the module-level parallelism available in C and Java programs could potentially be exploited using chip multiprocessors with relatively few processor cores. Again, there is not any big difference across C and Java programs.

4.4. Impact of thread-management overhead

Figure 6 shows speedup with overhead for speculation support. We have included three types of overhead: starting a new speculative thread, performing a rollback on misspeculation, and committing speculative state when a thread

has successfully finished. A thread that has been squashed as part of a rollback will incur a new thread start overhead when it is called again.

In the figure, the three types of overhead are set to the same size; we ran simulations for 10, 100, or 1000 cycles. For the sake of comparison, we repeat the speedup for the no-overhead machine. The numbers are for an 8-way machine.

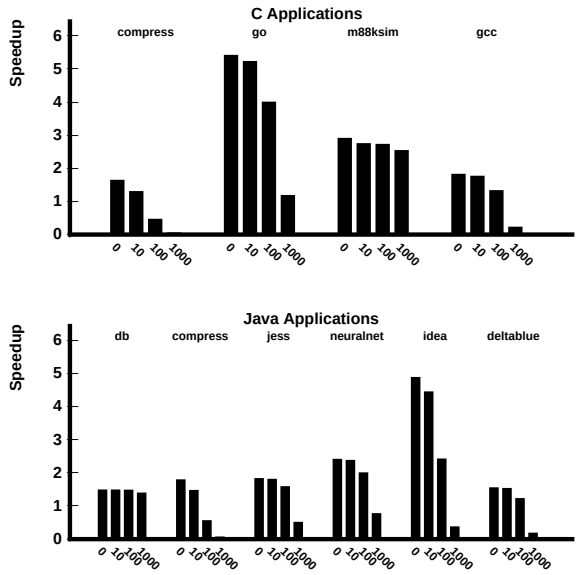


Figure 6. Speedup with thread-management overheads of 0, 10, 100 or 1000 cycles on an 8-way machine

The 100-cycle overhead simulations are interesting since they approximately correspond to the overheads reported for module speculation support in the Hydra CMP [6]. At a 100-cycle overhead, we can already see a severe impact on the speedup for several applications, even a slowdown for both compress programs. When the overhead is increased to 1000 cycles, the compress programs are more than ten times slower than their sequential execution.

In order to find the reason for this, we do not need to look further than module size. Figure 7 reveals that for both C and Java compress, the majority of the modules are shorter than 20 cycles, and almost all are under 100 cycles. This means that thread-management overheads will dominate execution time, since each module will, at least, give rise to a thread start overhead when called, and a commit when it reaches return. On the other hand, some of the modules are very large, which explains why the average sizes presented in Table 1 are several thousand instructions for some of the programs. Note that with our single-issue, perfect memory machine, there is a one-to-one correspondence between the number of cycles and instructions.

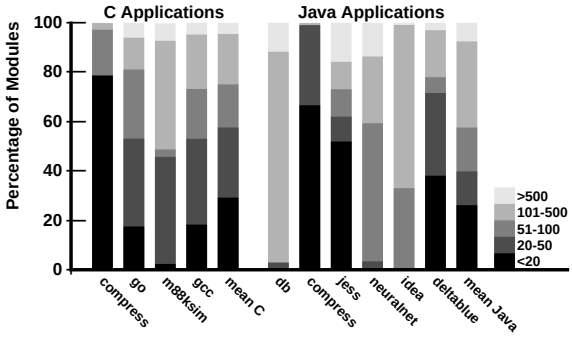


Figure 7. Percentage of modules (dynamic) of size <20, 20-50, 51-100, 101-500, or >500 cycles

We have observed that one of the side effects of increasing the number of processing elements is that the number of dependence violations will also increase. Therefore, with high thread overheads, the benefits of adding more processing elements will be smaller than indicated in Figure 5, in some cases we can even get a slowdown.

In Figure 8 we can see how the execution time is used. The execution time for a program in this figure is the total used time on all PEs added together. The simulations are run with 100-cycle thread-management overheads on eight PEs.

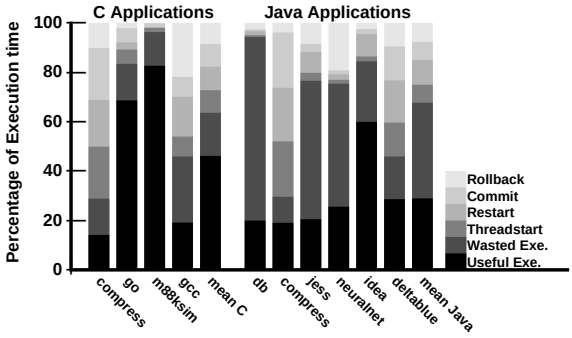


Figure 8. Part of execution time that is useful, wasted because of rollbacks, and used for thread-management. 100-cycle overheads on an 8-way machine

Useful execution is the part of the execution that was successful and committed; it is the part that corresponds to the sequential execution. Wasted time is the execution time that was thrown away because of a rollback or when the thread was squashed. Restart is the effect of additional thread start overhead for a thread that was squashed and

must be started again. The remaining three categories show thread start, rollback, and commit overhead.

This figure points out one of the serious disadvantages of speculative execution. Only 20% on average for Java programs, or 40% for C programs, of the processing time is useful execution. This is a disadvantage in a multitasking environment where other processes might make better use of the resources. It is also a problem from an energy-efficiency perspective. Wasted execution makes up a major part of the total processing time for most benchmarks. A conclusion would be that methods for minimizing the number of misspeculations, and thus wasted execution, is probably needed even if it is not necessary from the point of view of performance for a single-application.

It is clear from this experiment, however, that keeping overheads small is of utmost importance for module speculation support.

5. Related work

There is a large body of research in the recent literature that focuses on architectures and compilation techniques for speculative thread-level parallelism.

One of the first architecture proposals for thread-level speculation was done within the Multiscalar project [15]. One of the novel features of this architecture is the address-resolution buffer [4] that validates and signals violations to data dependences between threads. Another noticeable speculative architecture proposal is the superthreaded architecture [17]. Several distributed approaches to implement support for thread-level speculation have also been presented in the framework of chip-multiprocessors [5, 6, 16]. This study is based on the feasible inclusion of such a mechanism in chip multiprocessors.

Another important prerequisite for this study is the progress in value prediction done over the last few years. Value prediction enables speculation beyond the data flow limit. It was introduced by Lipasti *et al.* [7] as a way to hide memory load latency by allowing data dependent instructions to execute in parallel. The predictability of data values was investigated in [14]. Others have followed up with a number of inventive prediction schemes such as the stride and last value predictors [7] which we study in this paper.

This paper focuses on the opportunities and limitations of speculative module-level parallelism – a straight-forward method to extract thread-level parallelism out of existing software. Several recent papers have had similar goals. A limit study of the inherent loop-level as well as module-level parallelism in SPECint95 programs was recently published by Oplinger *et al.* [12]. While disregarding architectural limitations in terms of thread management overheads, they found that there is ample module-level parallelism in the benchmark suite that can be exploited by multiprocessor or multithreaded processor cores of typically less than eight processors. In comparison with our study, they didn't address how important memory-level dependences are and did not look at Java applications. Moreover, they didn't study how much typical overheads in CMP architecture models would affect the achievable speedup.

In contrast, Chen and Olukotun [2] focus on Java programs. Their study is mostly aimed at the speedup obtained on the Hydra CMP proposal and does neither address the impact of various value prediction schemes nor how scalable the performance is. While they note that thread management overhead may have a severe impact on the speedup, they didn't analyze how it relates to the size of the modules. In a follow-up study by the same team based on SPECint95 programs [11], they observe that thread-management overheads can be detrimental to the speedup obtained because of the penalties associated with misspeculations. As a remedy, they propose and evaluate schemes that select modules to speculate on depending on their likelihood to succeed.

Value prediction as a way to reduce dependence violations in thread-level data dependence speculation architectures has been investigated by Marcuello *et al.* [10, 9] in the context of their Clustered Speculative Multithreaded processor. They speculate on live input values to threads (values used but not defined within the thread) at thread start time. Some works mentioned earlier has also used value prediction for module return values [2, 6] and memory loads [12]. Others who have used value prediction in conjunction with coarse-grained speculative architectures include [13, 1, 3]. However, to the best of our knowledge, our study is the first to address the limits on value prediction which pin-points whether there is room for improvements.

6. Conclusions

The goal of this study has been to understand the impact of the programming style – imperative versus object-oriented – on the inherent module-level speculative parallelism as well as how architectural deficiencies in proposed chip-multiprocessor architectures affect the achievable speedup.

One would expect that object-oriented programs would make more heavy use of modules and would encapsulate many of the data dependences with a potential to expose more module-level parallelism. Contrary to this intuition, we found that there is not any significant difference between the inherent module-level parallelism in the C versus the Java programs that we studied. In both cases, we observed a speedup limit of about 3.5. In addition, the two suites representing the two programming styles were both sensitive to memory-level data dependences which suggests that progress in memory value prediction schemes are important to approach the maximum speedup. As for return-value pre-

diction schemes, simple ones based on last- or stride-value fare pretty well across all applications.

When considering the impact of architecture-level constraints, we found that all of the inherent parallelism could be exploited by typically small multithreaded or multiprocessor cores with less than eight processors. However, a key inhibitor to reaching the speedup limit is the overheads imposed by thread management including the time to start (or restart), commit, or roll-back threads upon data dependence violations. Given the fairly small module sizes, speedup is severely affected when the overhead exceeds a hundred cycles. This calls for more efficient thread management mechanisms than what is currently known in the literature for chip-multiprocessor architectures. Obviously, using MLP in more loosely coupled architectures is not an option.

In this study, we did not try to enforce a certain thread granularity, instead all modules were parallelized regardless of size. On realistic architectures, with various overheads, granularity is important. Methods to selectively apply module-level speculation would be needed. Our reason for using modules as the only source of parallelism was that they are control independent and easy to identify. Since the amount of MLP is limited, additional sources of parallelism are needed in order to achieve large performance gains using thread-level speculation techniques, but likely at the expense of increased complexity.

Acknowledgments

We would like to thank Peter Rundberg and Magnus Ekman of Chalmers University of Technology, as well as the anonymous reviewers, for their valuable comments on earlier drafts of this paper. This research has been financially supported by the ARTES/PAMP program of the Swedish Foundation for Strategic Research (SSF) and by equipment grants from Sun Microsystems Inc. and the Swedish Council for Planning and Coordination of Research (FRN) under contract number 96238.

References

[1] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st Annual International Symposium on Microarchitecture (MICRO '98)*, pages 226–236. IEEE Computer Society, Dec. 1998.

[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. Franklin and G. Sohi. Arb: A hardware mechanism for dynamic memory disambiguation. In *IEEE Transactions on Computers Vol. 45 No. 5*, pages 552–571. IEEE Computer Society, May 1996.

[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] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII '96)*, pages 138–147. ACM Press, Oct. 1996.

[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. 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.

[10] P. Marcuello, J. Tubella, and A. Gonzalez. Value prediction for speculative multithreaded architectures. In *Proceedings. 32nd Annual International Symposium on Microarchitecture (MICRO '99)*, pages 230–237. IEEE Computer Society, Dec. 1999.

[11] K. Olukotun, L. Hammond, and M. Willey. Improving the performance of speculatively parallel applications on the hydra CMP. In *Proceedings of the 1999 International Conference on Supercomputing (ICS '99)*, pages 21–30. ACM Press, June 1999.

[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] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO '97)*, pages 138–148. IEEE Computer Society, Dec. 1997.

[14] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO '97)*, pages 248–258. IEEE Computer Society, Dec. 1997.

[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.

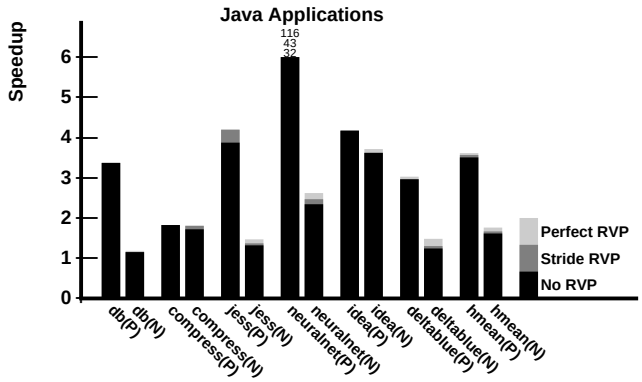
[16] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA '98)*, pages 2–13. IEEE Computer Society, Feb. 1998.

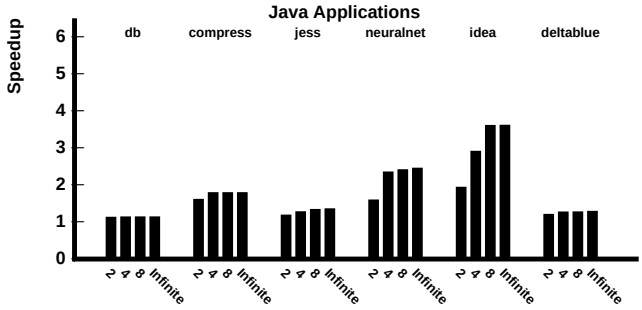
[17] J.-Y. Tsai and P.-C. Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 35–46. IEEE Computer Society, Oct. 1996.

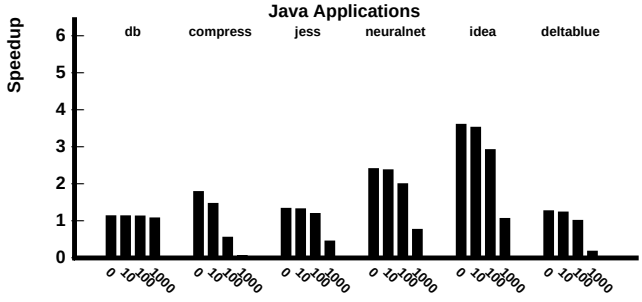
Errata

Some results reported in the paper *Limits on Module-level Parallelism in Imperative and Object-oriented Programs on CMP Platforms* are incorrect due to a bug in the thread-level speculation simulator. The affected results are for db(N), jess(N) and deltablue(N) in Figure 4, and db, jess, deltablue, and idea in Figure 5 and Figure 6. These programs have worse speedup than reported in the paper; however, the errors do not affect the qualitative conclusions in the paper.

Correct speedup values for the Java applications are shown in the updated figures below (none of the C programs were affected). From top to bottom: value prediction (Figure 4), limited number of processors (Figure 5) and thread-management overhead (Figure 6).







Paper II

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

Reprinted from

Proceedings of the International
Parallel and Distributed Processing Symposium (IPDPS 2003),
April 2003.

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-

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

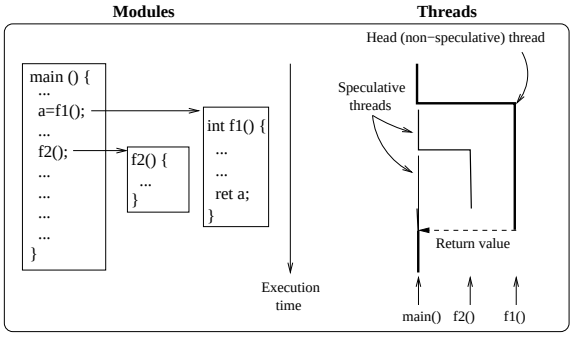


Figure 1. Execution model. New speculative threads are spawned for module continuations.

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.

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 (dynamic)*	*#Modules (dynamic)*	*Avg. instr./mod. (dynamic)*	*#Modules (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-

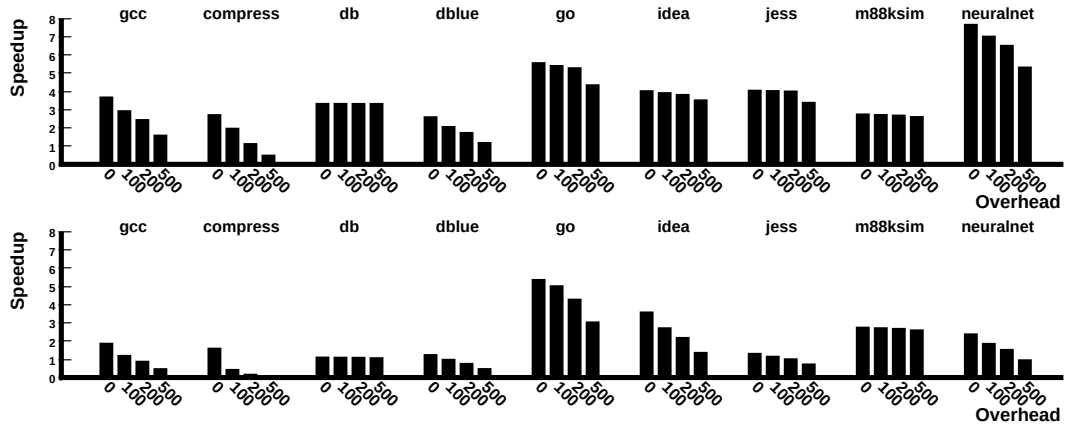


Figure 2. Speedup with thread-management overhead 0-500. The graphs show results with perfect (upper) and realistic (lower) value prediction models.

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.

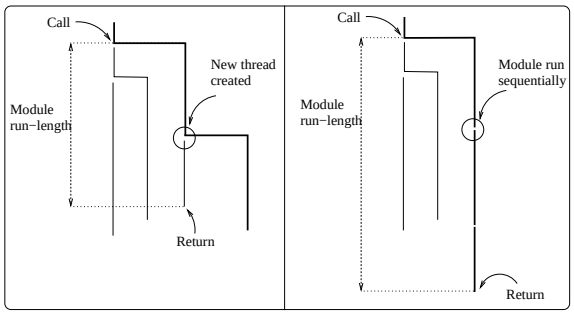


Figure 3. Module run-length calculation.

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

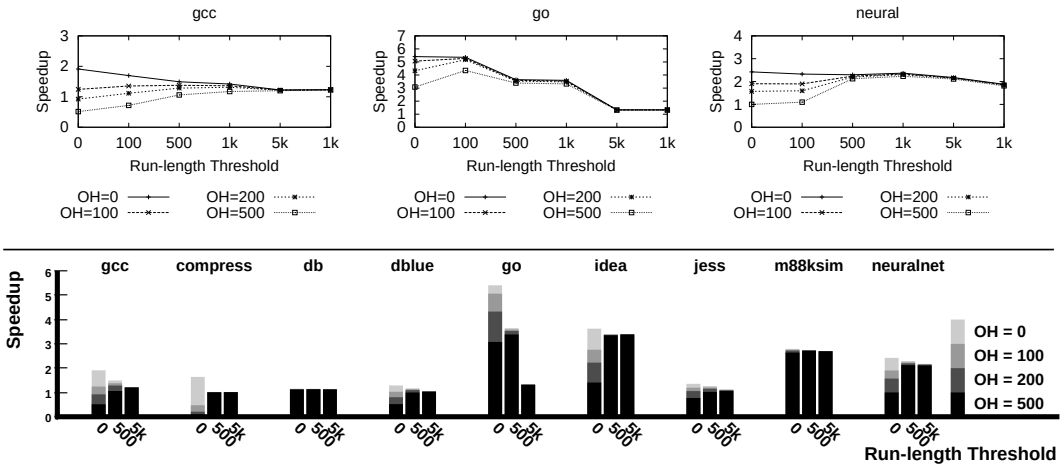


Figure 4. Speedup with module run-length thresholds between 0 and 10000 cycles.

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

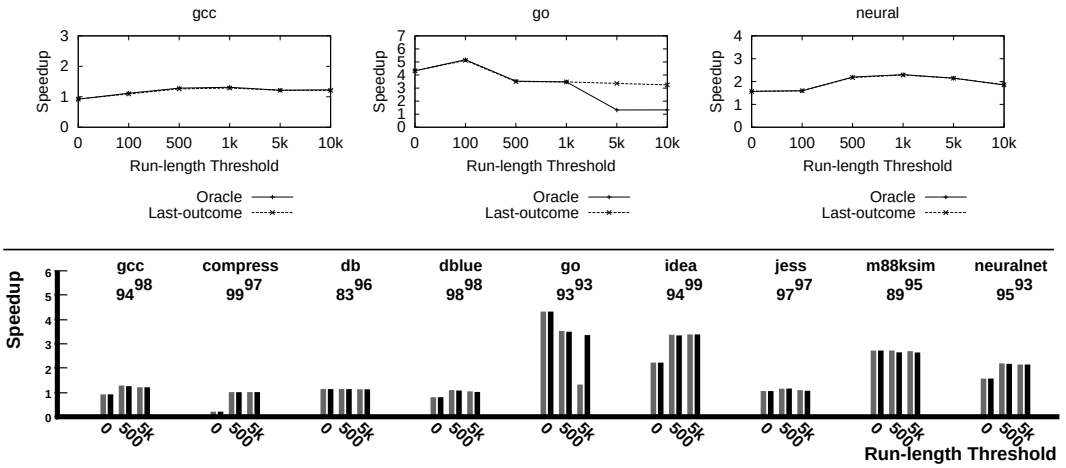


Figure 5. Speedup of the last-outcome run-length predictor (black bars) compared to the ideal predictor (grey bars), with 200-cycle overheads. Prediction accuracy (in %) is printed on top of the bars.

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 run-

ning 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.

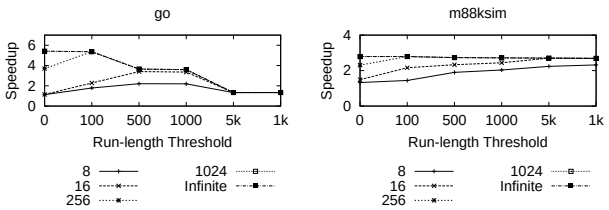


Figure 6. Benefit of run-length thresholds with limited live threads.

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.

Paper III

Reducing Misspeculation Overhead for Module-Level Speculative Execution

Reprinted from

Technical Report 03–07
Department of Computer Engineering
Chalmers University of Technology
Göteborg, Sweden, April 2003.

Submitted for publication.

Reducing Misspeculation Overhead for Module-Level Speculative Execution

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

Abstract

Thread-level speculative execution is a means to widen the scope of single-threaded applications that can perform well on chip-multiprocessors. We consider module-level speculation, i.e., speculative threads execute the code after a module (i.e., a procedure, function, or method) is called. Unfortunately, previous studies have shown that indiscriminate speculation results in significant overheads due to misspeculation that can wipe out the gains from parallelism.

In this paper, we present and evaluate in detail several techniques that aim at reducing the overhead associated with misspeculations. We evaluate how much one can gain by saving the work executed prior to a misspeculation by using more aggressive checkpointing. Based on nine Java and C applications we find that the savings can be quite high in contrast to what previous studies that considered loop-level parallelism showed. We then propose and evaluate a number of methods that aim at selectively, on a per-module basis, only allow speculation when the likelihood of a misspeculation is small. The best method is shown to reduce the overhead from 336% over a serial execution down to 54% on average. In addition, the speedup for half of the applications is improved in this process.

Keywords: Multiprocessors, chip-multiprocessors, thread-level speculation, module-level parallelism, misspeculation prediction, performance evaluation.

1 Introduction

Chip-multiprocessors (CMPs) are an emerging architectural style owing to the diminishing returns in parallelism exploitation and the increased complexity of the superscalar paradigm. CMPs trade thread-level parallelism for the instruction-level parallelism provided by the superscalar processor cores. Unfortunately, most applications are single-threaded and are thus a bad target for CMPs unless they can be automatically parallelized.

Speculative thread-level parallelism (STLP) paradigm 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 dependences are resolved. Since chip-multiprocessors have a tight integration with low latency and high bandwidth communication between the processor cores, they are a good match for STLP. Consequently, there is a multitude of proposals for integrating a speculation system with a chip-multiprocessor [2, 4, 5, 7, 10, 11, 14, 15].

There are several approaches to create threads for the STLP machine, for instance at loops or module (i.e. procedure, function, or method) invocations. Since one of our main goals is to explore techniques that can be used for run-time parallelization of unmodified binaries, module-level parallelism (MLP) is an appealing option: modules are easy to identify also at run-time.

The drawback with too aggressive module-level speculation, i.e. running all available modules speculatively, 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 the sequential execution. Overhead is defined as all extra execution associated with events that do not occur in the sequential execution of the program. Obviously, this includes all extra work the speculation system performs in order to manage speculative threads. The *thread-management overhead* consists of thread-start, roll-back, and commit (thread completion) overhead.

Some overhead is compulsory and its impact is proportional to the size of the overhead compared to the size of the speculative threads – thread-start and commit belongs to this category. The impact can be kept under control with efficient speculation mechanisms and by avoiding to create too small threads. In [17] we presented a method that will prevent small modules from being used for speculation.

The other category is overhead caused by misspeculations and its impact is more difficult to predict. A misspeculation triggers the roll-back mechanism, which will squash erroneous threads and bring back execution to a correct state. In addition to the roll-back handler overhead, the partially completed threads that were squashed need to be re-executed. Apart from the lost parallelism, this *execution overhead* is unwanted for several reasons. First, threads that are squashed will have occupied processing resources, communication, and storage space for its speculative state without contributing to the forward progress of the program; potentially hampering successful threads, and slowing down execution. Second, even if there are plenty of free resources, execution

overhead is a serious drawback when considering energy-efficiency or the performance of other programs in a multiprogrammed environment.

In this paper, we investigate two methods that can be integrated in the speculation run-time system, which selectively disables speculation where spawning a new speculative thread is expected to do more harm than good. Our goal is two-fold: we want to limit the execution overhead, and if possible improve speedup, compared to naively spawning new speculative threads for all module calls. In addition, we look at how the amount of overhead and speedup is affected by the method for roll-back used by the speculation system.

We investigate two methods for roll-back: The first is *thread roll-back*, where the entire thread as well as its child threads are squashed if it suffers from a dependence violation. The other method, which we call *perfect roll-back*, can restart execution from the violating instruction, thus saving work performed before this instruction in the violating thread. Olukotun et al. [9] implemented a checkpointing mechanism that can be used for such partial roll-backs. While they did not see a significant gain with their loop-level parallelism, we find that with module-level parallelism there are applications with a performance gap between the two methods, and that a checkpointing mechanism indeed would be useful provided that we can find the right locations to save checkpoints.

We then propose and evaluate several techniques that aim at selectively predict which modules to speculate based on the likelihood that they will cause misspeculations. The first method we consider is to *predict parallel overlap*, or the time a thread and its speculative child executes in parallel, in order to disable speculation when there is small or no parallelism to gain from spawning a new thread. While the profiling run shows that seven of nine applications improve speedup, a real predictor does not reach the same success. The overhead is down to a mean 158% from an original 336%, but is still significant.

With the second technique, *misspeculation prediction* we try to avoid creating threads that will misspeculate by recording such events, and use history-based prediction when deciding whether to create a new thread or not. 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 average overhead to 54%, while improving speedup for four applications, but with noticeable worse speedup for two applications.

The applications that do not benefit from misspeculation prediction are those who suffer from fewer misspeculations to begin with. By applying misspeculation prediction selectively, i.e. only when the ratio of squashes compared to new thread starts is above 0.6, we avoid the worse speedup for the applications with few misspeculations, but at the price of a somewhat higher 89% mean overhead. However, this method gives the same or slightly higher speedup for all applications as running all modules speculatively, but with almost four times lower average overhead.

We begin by explaining the execution model and simulation tools in Section 2. Section 3 investigates the performance difference between perfect and thread roll-back.

Then, sections 4 and 5 present the parallel overlap and misspeculation prediction techniques, respectively. In Section 6 we investigate a number of predictors for the misspeculation prediction technique, and misspeculation prediction is improved upon with a method for selective use in Section 7. Related works are discussed in Section 8, and finally, we conclude in Section 9.

2 Methodology

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

2.1 Execution Model

In the thread-level speculation model, several threads derived from the same sequential code are executed in parallel. As opposed to other parallelization techniques, there are no guarantees that the threads are independent; instead, a *speculation system* detects and corrects errors caused by data dependences – misspeculations.

The threads are ordered after their position in the sequential code, a thread derived from a piece of code in a sequential execution of a program is said to be less speculative than all threads derived from code that comes after it in the same sequential execution. A misspeculation occurs if there is a flow (or read-after-write) dependency between two threads and the more speculative thread reads the shared variable before the less speculative thread has written it. Other types of dependences (anti-, output-) are resolved through renaming by the speculation system.

The speculation system takes care of spawning new threads, checking for data dependences between these 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 the results in special-purpose buffers or in a modified cache-hierarchy until the thread is non-speculative (i.e. there are no less speculative threads in the system). At that point, the thread can commit; that is, results are merged with main memory state and the thread is removed from the speculation system.

Module-level thread speculation treats module calls as potential points used to spawn a new speculative thread. When encountering a call instruction, the original thread will continue to execute the called function and, if the call 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.

Implementations of module-level speculation have earlier been described in detail

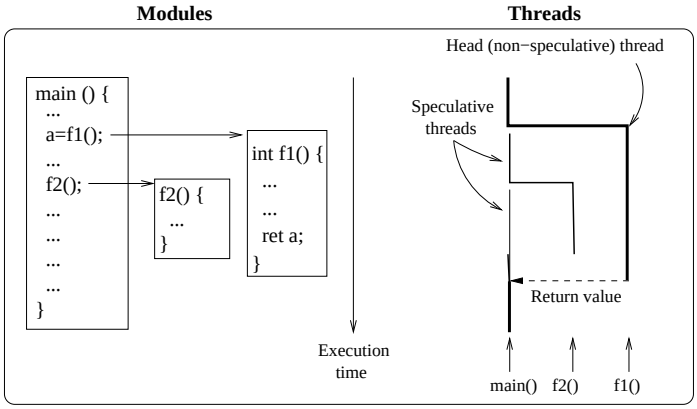


Figure 1. Execution model. New speculative threads are spawned for module continuations.

by Steffan et al. [12] and Hammond et al. [4] for their respective chip-multiprocessor speculation systems.

2.2 Simulated Machine

Our simulated architecture is an eight-processor machine with support for speculative thread-level parallelism. The model is kept relatively simple; we focus on the impact of misspeculation overhead on the inherent module-level parallelism. This has allowed us to make many simplifying assumptions as to the level of detail of the architecture model. For example, we disregard the impact of communication latencies, contention, and out-of-order execution. The following is a summary of our system parameters:

- All communication and memory accesses are completed in a single cycle; the processor issue one instruction per cycle in order.
- We impose a fixed-length 100-cycle overhead for thread-starts, roll-backs, and context switches. This conforms with the experiences in the Hydra project[4]. Dependence checking and to some extent commit can be performed efficiently with hardware support, therefore we do not model overhead for these events. For instance, the Speculative Versioning Cache [3] does dependence checking and lazy commit of speculative state in the memory system as an extension to the cache coherence protocol. Commit will include some clean-up overhead but is much less critical than thread-start and roll-back since it only happens once per thread – when it is known to have finished correctly – and is not on the critical path of the application.
- The speculative threads can be preempted; the scheduling policy will make sure that the N least speculative threads are always running, where N is the number of processors. (We assume N=8 in the experiments.) However, there can be as many threads as necessary active in the speculation system.

- The speculation system resolves anti- and output dependences as well as forward values between threads in order to avoid flow dependence violations whenever possible. Remaining dependences cause a roll-back. We look at the roll-back policy in Section 3.
- Stride value prediction is used on module return values in order to remove easily avoidable dependences such as functions always returning the same value (it could be, for instance, an error code which almost always is zero) or a constant stride value (for instance, a sequence number). In a previous study [16], we found that this prediction method was very effective in eliminating flow dependence violations based on return values.

2.3 Simulation Setup

A high-level view of the experimental approach is shown in Figure 2. We first annotate the program at each module call and return, and at each point 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.

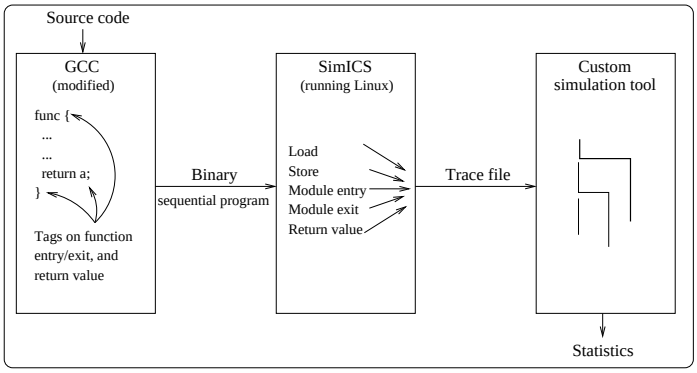


Figure 2. The GCC, Simics, and STLP simulator tool-chain.

The binaries are run on a Linux system on top of a SPARC-based instruction-set simulator using Simics [6]. 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.

Finally, the trace is fed into a simulation tool (called Custom simulation tool in Figure 2). Based on the sequential trace, this tool models the simulated architecture; on a module invocation a new thread is spawned and scheduled on a new processor, if available. It uses the events and time-stamps in the trace to simulate program execution with thread creation and completion, memory accesses and dependence checking, and roll-backs. This simulator also implements the techniques investigated in this paper, and outputs speedup results, overhead count, and other useful execution statistics.

Our version of Simics simulates a single-issue in-order SPARC V8 architecture. In

Table 1. Benchmark applications.

Name	*Origin*	*Description*	*#Instr. (dyn.)*	*#Modules (dynamic)*	*Avg. instr/mod (dynamic)*	*#Mod. (static)*
gcc	SPECint95	C Compiler	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

addition, we use a perfect (1 cycle-access) memory system. Using superscalar architectures and realistic memory and communication system would introduce additional effects, for instance, inter-thread communication and memory-access latencies, bus contention, and a possible trade-off between thread-level and instruction-level parallelism. While these are important issues to investigate for a real implementation of thread-level speculation in a chip-multiprocessor, we have so far chosen a simpler machine model in order to be able to study thread-management and misspeculation effects in isolation on the inherent module-level parallelism without interference from such system-dependent parameters.

2.4 Benchmarks

Table 1 summarizes the benchmark applications. We have used the same applications as in our previous studies [16, 17]. We use three applications from SPECint95, two from SPEC JVM98 and jBYTEmark each, and one application from Sun Labs. The input data sets are quite small, but the number of modules in each trace is still considerable as shown in Table 1.

We have concentrated on integer programs known to be hard to parallelize with parallelizing compilers, and that 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 require a tightly integrated STLP machine with low communication latencies; a chip-multiprocessor or multithreaded CPU.

3 Significance of Roll-back Policy

One reason why roll-backs are especially harmful is because in addition to the overhead of the roll-back handler code, useful work is being thrown away. How much work is thrown away depends on how accurately the roll-back mechanism can squash work affected by the violation. An optimal mechanism would only squash and re-execute instructions that depend on the erroneous value. However, this would entail tracking the effects of the erroneous value through the violating and dependent threads, which is not practically feasible.

The mechanism which is most straight-forward to implement, and commonly suggested for STLP machines, is to squash the entire thread containing the violating instruction, as well as any thread spawned from this thread. We call this method *thread roll-back*, since the entire violating thread is squashed.

Another method, which we have used in previous studies [16, 17], is to re-execute everything after the violation, but save work performed by the thread prior to the violation. This method would require checkpoints before every exposed load and used return value, so that the roll-back mechanism could restore the correct state at any such event. The difference between thread roll-back and this method, which we call *perfect roll-back*, is visualized in Figure 3. At each checkpoint, the register contents must be backed up and the speculative memory state after the checkpoint must be separated from the state prior to the checkpoint. A checkpoint mechanism was described by Olukotun et al. in [9].

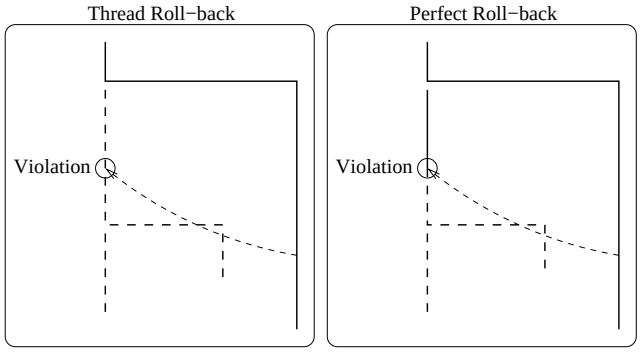


Figure 3. The violation causes the dotted part of the thread to be squashed.

Since checkpointing requires extra time and resources, it could be done on a select few loads instead of every single one. In [4], checkpoints are saved before violation-prone loads. Another alternative could be to checkpoint before a load after a certain amount of instructions have been executed since the previous checkpoint, in order to decrease the average roll-back distance.

Figure 4 shows the performance difference between thread roll-back and perfect roll-back. For each application, the height of the bars show speedup on the baseline speculative machine, containing eight processors, compared to sequential execution. The grey bars show speedup with perfect roll-back, and the black bars are for thread roll-back. The numbers on top of the speedup bars show the total (thread-management and execution) overhead as the percentage of extra processor cycles, compared to the sequential execution, that were used in the speculative execution of the program. That is, for the parallelized program, the cycles from all used processors are added together.

As we can see, some of the programs, notably Go and Idea, suffer from the less precise thread roll-back; in total, five of the applications have notably lower speedup.

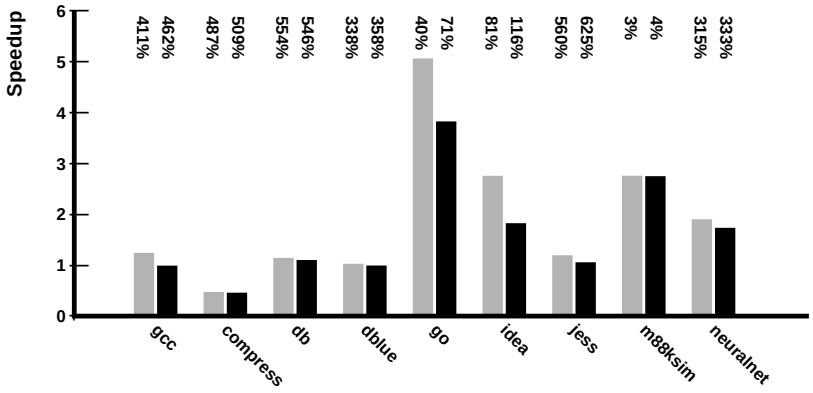


Figure 4. Performance of perfect (grey) vs. thread (black) roll-back.

In addition to the lower speedup, the total overhead increases for all applications but Db. On average the overhead is 336% with thread roll-back compared to 309% for perfect roll-back. This increase is due to the fact that more partially completed work is squashed and re-executed for each misspeculation. As opposed to the results reported in [9], which focused on loop-level parallelism, we can see that the coarser thread roll-back can have a detrimental impact on the speedup of our programs. If we can find a way to identify where checkpoints should be saved, a checkpoint mechanism would indeed be useful for module-level parallelism.

Because of the lower implementation complexity of thread roll-back, however, we will assume that type of roll-back in the rest of the paper.

4 Predicting Parallel Overlap

Threads that roll back well into their execution are the worst offenders, since more work is squashed the longer the thread has executed, and the possibility to still do some useful parallel work decreases. The rationale behind this method is that if the *parallel overlap* – the time a thread and its child executes simultaneously – is sufficiently small, it is either because the module was small, *or* because the child was recently restarted after a misspeculation. Therefore, we apply a minimum threshold for the overlap; calls where the overlap is found to fall below the threshold are classified as non-parallel and we use prediction in an attempt to avoid creating a new thread at these calls for future invocations.

If the parallel overlap is smaller than the thread-start time, spawning a new thread will not contribute any useful parallel work. As a result, it makes sense to have a threshold at least as large as the thread-start overhead, which will weed out both misspeculating and too small modules. Larger thresholds may be useful if the misspeculating modules are the ones disabled, but useful parallelism can be removed in the process.

An advantage of this technique is that threads that suffer from a misspeculation early in their execution are not classified as non-parallel. If a thread rolls back early, less

work has been wasted, and there is a good chance that it will still contribute with useful parallel work. The potential drawback is that the total overhead can still be large, since these early misspeculations are allowed.

This technique is related to run-length prediction, which we presented in [17], in that a threshold is used to assess the parallel overlap, but the way overlap is measured and used differs. Run-length prediction measures the length of modules and predicts them to be non-parallel if the length is below a threshold. The module run-length includes modules run sequentially within a module, but excludes all overhead and idle (preempted) time. The aim of run-length prediction is to find threads of suitable length for speculation, not to reduce misspeculation. A big advantage is that the run-length can be measured even when speculation is disabled, so the technique can easily adapt to changing circumstances during execution.

The parallel overlap technique is affected by misspeculations, which is important as we are trying to reduce overhead. It is also simpler to implement; however, once the no-speculate prediction has been made, we can not measure if the overlap remains below the threshold, since future module invocations will run sequentially.

4.1 Algorithm & Implementation

Figure 5 shows how the parallel overlap is measured. When a thread (T1) completes its execution, the start or latest restart time of the most recently spawned child thread (T2) is checked and the difference between the child start time and current time is the execution overlap between these two threads.

If the start time is far enough in the past, as in 5 (a), the execution overlap will exceed the threshold and the speculation is classified as successful. If, on the other hand, the thread is squashed and restarted late as in Figure 5 (b), the execution overlap falls below the threshold, and we classify the child thread as undesirable. When this happens, a prediction mechanism is used that aim at preventing the same thing from happening again. Because of the simple way to measure overlap, this technique is only suitable for thread roll-back where the whole thread is restarted after a misspeculation.

Implementing this technique requires bookkeeping of the start time of each active thread, as well as a table that predicts whether or not to speculate. We store one prediction per module, and index the prediction table with a module ID. The ID could be any identifier unique for the module, such as a sequence number or the address of the first instruction. As is shown in the rightmost column in Table 1 there are only up to a few hundred modules in the applications. Therefore, if we store one prediction per module, the prediction table does not need to be larger than a few hundred bytes to avoid having several modules map to the same slot. In the simulations we assume an infinite prediction table, which is not unreasonable due to the small number of modules. If the table is stored in the memory hierarchy, it is automatically shared between the processors.

The extra work needed for this algorithm consists of:

- Record start time for each thread. The speculation system needs to keep a list

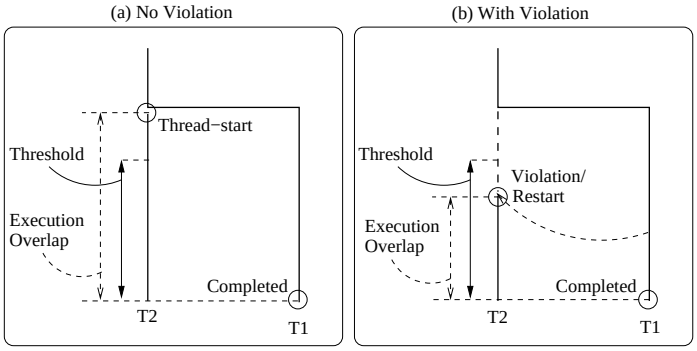


Figure 5. Calls are marked as non-parallel if the parallel overlap is below the threshold.

of active threads; the start time – this can be a cycle count obtained from the processors' built-in performance counters – is recorded at thread start.

- When a call instruction is encountered, the prediction table is accessed; if the prediction is *speculate* a new thread is created, otherwise the speculation system does nothing. If the start address of the module is used as module ID the index to the prediction table is the same as the target address of the call instruction.
- When a thread is completed, the start time of the closest more speculative thread is read and compared to the current time; if the difference is below the threshold a *no-speculate* prediction is written to the prediction table, otherwise nothing needs to be done.

These operations can be added to thread-start and completion operations in the speculation system with only a few extra instructions and memory accesses – which is not much compared to a full thread-start. Since the overhead for the overlap prediction operations is expected to be small compared to the existing overheads, we add no extra time for them in our simulations.

4.2 Experimental Results

In order to evaluate the potential of predicting parallel overlap, we first run simulations with profiling; the application is executed once while call instructions where overlap is below the threshold are marked as non-parallel. The same workload is then re-executed with speculation disabled for these calls. Note that this does not necessarily give an exact result according to our definition of parallel overlap – as soon as one thread is removed the rest of the execution will change, which is not taken into account in the profiling run – but it will give us a fair estimate of the effectiveness of the method.

Figure 6 shows speedup results and total overhead for the applications with overlap thresholds of 100, 150, and 200 cycles. The lowest threshold, 100, is equal to the thread-start overhead, and will therefore remove threads that do not overlap enough to account for the thread-start procedure. As a reference, results for thread roll-back from

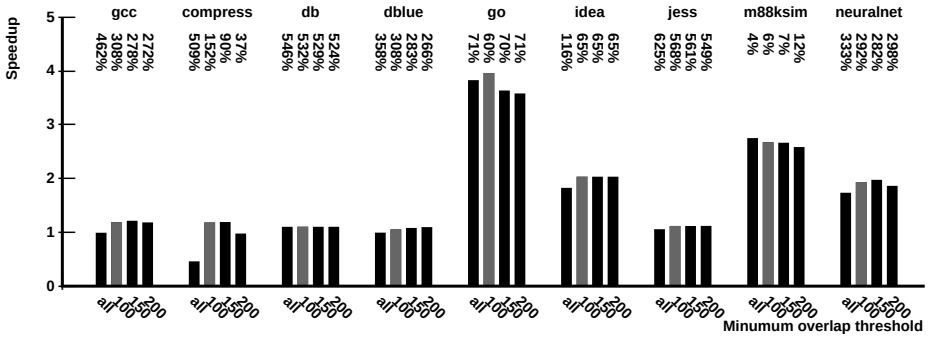


Figure 6. Disable speculation if overlap is less than 100, 150, or 200 cycles.

Figure 4 are included, they are labeled *all* since all calls will spawn a new thread.

An overlap of 200 cycles does not seem to be very much, but we can already see how speedup starts to decline for several of the benchmarks (Gcc, Compress, Go, M88ksim, Neuralnet), which suggests that higher thresholds are not useful. On the other hand, a threshold of 100-150 is beneficial for most of the programs; only Db shows virtually no improvement, and M88ksim a small decrease in speedup (M88ksim is an exception, where using this technique happened to cause new dependence violations and increased squashing). However, the drawback of the technique is that the total overhead is still large for many of the programs. For instance, Gcc requires almost three times as many clock cycles as the sequential execution in order to achieve a speedup of about 25%.

The threshold that yields the best speedups, highlighted in grey, is the one equal to the thread-start overhead of 100 cycles. For this threshold, the average overhead for all applications is down to 255% compared to 336% when starting all modules speculatively (the leftmost bar).

Figure 7 shows the all speculative (right), and profiling (grey) results carried over from Figure 6, compared against new results from a predictor working in run-time. It is a per-module, last-outcome predictor updated at thread completion. Both profiling and predictor results are with a 100-cycle overlap threshold.

Overall, the predictor is somewhat over-zealous in disabling speculation compared to the profiling run, which can be seen on the lower numbers for total overhead (mean 158%). This translates into lower speedup for Compress, Go, and M88ksim, slightly lower for Db and Jess, but slightly better for Gcc and Neuralnet. Especially for Go, however, the predictor is not performing well, and the overhead is still significant in many of the applications.

The results show that while the simple predictor manages to reduce the overhead, this is to the expense of losing parallelism exploitation opportunities in comparison with the profiling method. In addition, the overhead remains high; on average 255% and 158% for the profiler and the predictor, respectively, with five of the applications having in

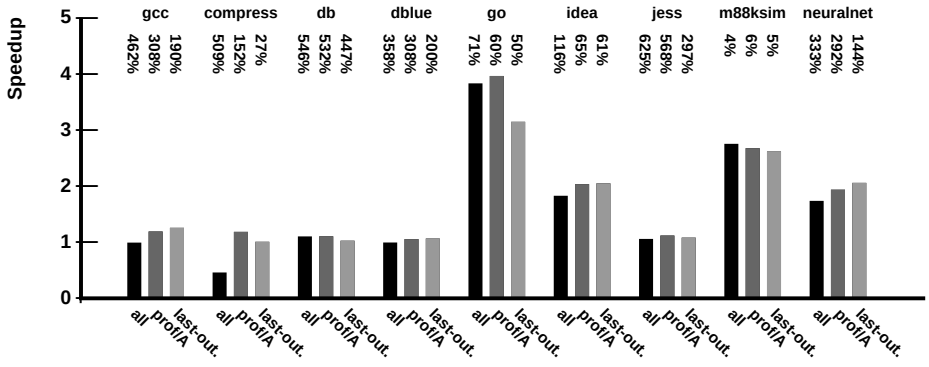


Figure 7. Last-outcome parallel overlap prediction compared to the profiling results.

excess of 200% overhead. The speedup is improved for seven of the applications in the profiling run, but only five with the predictor, and three applications performed worse using the technique. In the next section, a more effective method is presented.

5 Predicting Misspeculations

The second technique is more aggressively targeting misspeculating threads in order to bring down the overhead. We attempt to selectively disable speculation whenever a thread causes frequent misspeculations. Every time there is a dependence violation, the call tree is analyzed and we try to identify a confluence point where, if it is not used for speculation, the dependence violation will disappear. As with the previous technique, we then propose to use prediction in order to avoid expected misspeculations when the same situation occurs again.

The main advantage with this technique is that all misspeculating threads are targeted while the successful ones are left alone. This ought to ensure low overhead provided we are successful in implementing a predictor for the method. A drawback is that also threads that misspeculate early and later contribute with useful parallelism are affected; we do not search for the best trade-off for maximum speedup.

5.1 Algorithm & Implementation

A first concern is which call instruction to mark 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 8, there is a dependence violation between a store in thread T1 and a load in T4 (higher numbered threads are more speculative). 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.

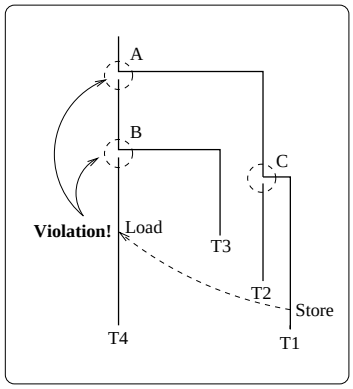


Figure 8. Finding calls to classify as non-parallel after a violation.

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, but a good heuristic would be to select *A*, the common ancestor for the threads involved in the violation, as it is 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 roll-back operation.

Only *A* and *C* together is 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 a prediction table similar to the parallel overlap technique. 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, and update the prediction table. 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 again, the overhead is small compared to the existing mechanism.

5.2 Experimental Results

As with the parallel overlap technique, 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; once again we

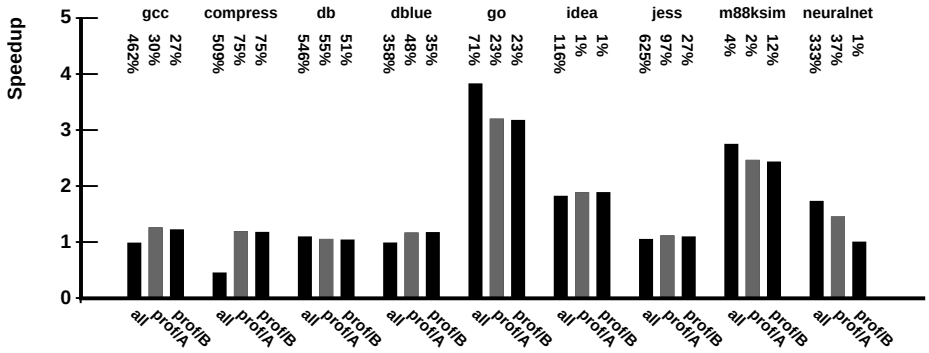


Figure 9. Profiling results for disabling speculation based on misspeculations, type A and B.

use an approximate profiling method.

In Figure 9, the leftmost bar marked *all* is again the naive implementation of running everything speculatively. The next two bars, *prof/A*, and *prof/B* show the profiling runs for type A and B respectively.

As we expected, the total overhead is greatly reduced. The average 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 overhead figure for B depend on two programs, Jess and Neuralnet. However, the speedup suffers, especially for Neuralnet which is all but serialized with *prof/B*. For this reason, 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 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.

6 Design Space for Misspeculation Predictors

The misspeculation prediction technique shows promise in bringing down the overhead. Therefore, we will make a more thorough investigation of misspeculation predictors than what we did with the parallel overlap predictor. We begin with a discussion of the design space and then analyze their performance tradeoffs.

6.1 Predictors & Implementation

Once we have identified unwanted threads we want to be able to predict and prevent future misspeculations. We use a table of predictors, but the scope of a predictor as well as size of the prediction table is affected by what we record in each *predictor entry*. Some interesting options how to index the prediction table are:

- Per-call prediction: Tie the predictor to the call instruction address; the predictor 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 predictor applies regardless from where the module was called.

The advantage with the policies having a broader scope are less storage space for the prediction table, and faster warm-up time. However, since more cases are covered by each predictor there may be cases where interference cause lower accuracy. 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. Again, 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 a *n*-bit predictor. We have run experiments with last-outcome and 2-bit predictors.

As mentioned, a disadvantage of both misspeculation and parallel overlap prediction 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. 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 predictor time out and reset to zero after it has been accessed *k* times.

The implementation issues are the same as those described in the previous section. The caller+callee indexing method will result in more predictors and require a larger table in order to avoid conflicts. The predictions use one or two bits each, plus a prediction expiration counter if we use timeout. For the ranges of interest, a six-bit timeout counter would suffice. Again, our simulations are with infinite prediction table size and no extra overhead added.

6.2 Experimental Results

In Figure 10, 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 in the prediction table. The *all* and *prof/A* results are carried over from the previous section for reference. 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.

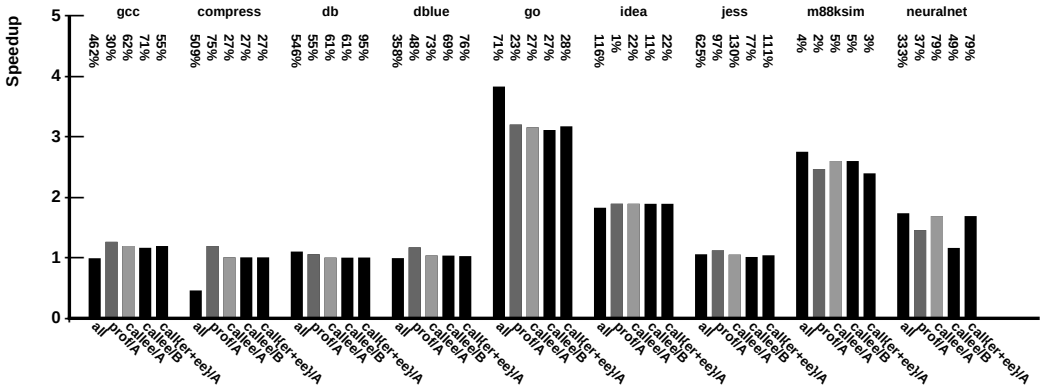


Figure 10. Comparison of misspeculation prediction policies.

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 run more modules speculatively than the profiler, with Compress being the exception. Not all misspeculating modules are correctly predicted as such. Therefore, the predictor show 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.

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 predictor with timeout. The results are shown in Figure 11. 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.

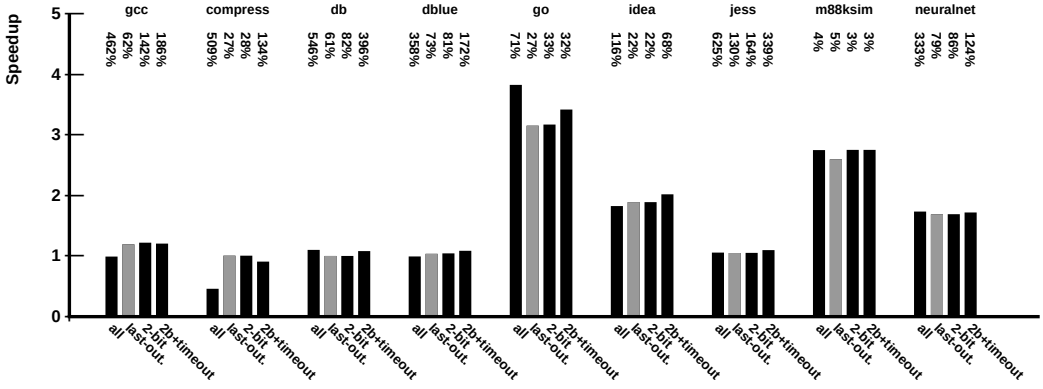


Figure 11. Performance of the last-outcome, 2-bit and 2-bit + timeout predictors.

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%.

In summary, the last-outcome predictor with the *Callee/A* predictor table, highlighted in grey in Figure 11, seems like 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.

However, a couple of the programs, Go and m88ksim, works better without the technique enabled at all. The overhead is relatively 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.

7 A Metric for 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, a couple of applications, 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 on 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 predictors are not used.

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

7.2 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 12, 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 increased to allow more misspeculations before the predictors are used, but there is yet no change in speedup.

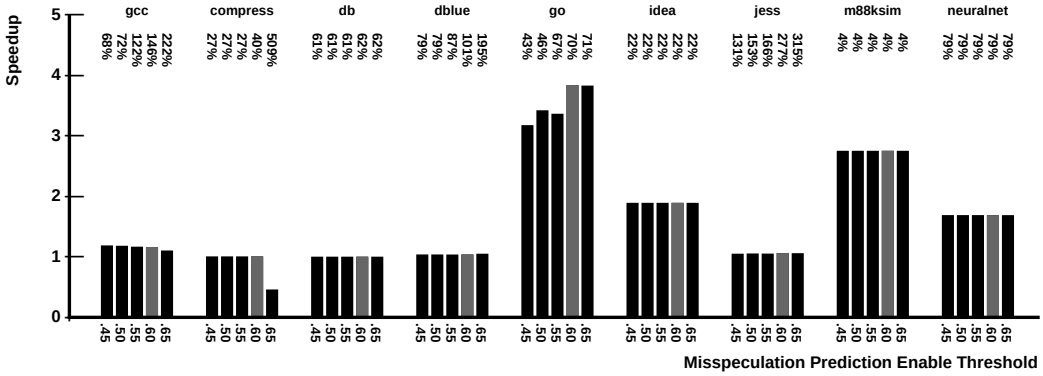


Figure 12. Threshold for misspeculation prediction.

The sensitive applications are Go and Compress. When the threshold is at 0.65, the speedup for Go 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.

8 Related Work

Hammond et al. [4] uses violation counters, thread timers, and stall timers to find and record non-parallel threads in a hardware prediction table. The violation counters are used to eliminate threads with many dependences. While their scheme is similar to our, they have not evaluated it in isolation, or explained the implementation in detail.

An alternative method to prevent speculation of dependent threads 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 [8], speculative chip-multiprocessors [13] and larger DSM machines [1] with some success. For the multiscalar processor, load-store pairs that are predicted to cause a violation are inserted in a synchronization table. Using this table, problematic loads are stalled until the corresponding store has completed and the value can be forwarded. The technique described by Steffan et al. [13] 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 [1] 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. The paper by Hammond presents 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 we also avoid unnecessary thread-start overhead.

Olukotun et al. [9] investigated using hardware checkpointing to bridge the gap between perfect and thread roll-back. The checkpointing mechanism works by separating speculative state before and after the checkpoint, as well as saving the register contents at the checkpoint. That way, roll-back to the most recent checkpoint prior to the violation is possible. They found only a few percent improvements with this technique; not enough to warrant the extra hardware. Our results show a significant gap between thread- and perfect roll-back for some applications. Unlike the study by Olukotun et al., we have not investigated how to use checkpointing selectively to bridge this speedup gap, but as we saw in Section 3 there is a potential gain of using checkpointing in module-level speculation.

9 Conclusions

When aggressively spawning speculative threads at all module invocations, the execution is dominated by overhead, both due to thread-management and re-execution of code after thread squashes. In our benchmark applications, we have found that the average overhead was 336% when running all modules speculatively.

The techniques presented in this paper are aimed at bringing down this overhead in order to save processing/communication resources as well as bringing down the extra energy used because of thread-level speculation. The techniques are aimed for inclusion in the run-time system and do not require recompilation of the programs.

A number of roll-back and misspeculation prevention techniques, as well as several misspeculation predictor designs were investigated. Overall, our contributions are the following:

- When targeting module-level parallelism, there is a potential gain in more aggressive checkpointing so that threads can be partially rolled back instead of entirely squashed. Five of the nine benchmarks have notably lower speedup with thread roll-back than perfect roll-back.
- The overhead with thread roll-back can be reduced from an average of 336% to 54% using a misspeculation prediction method where a last-outcome predictor is stored for each module. The predictor informs the speculation system whether to start a new thread or not when this module is called. However, the speedup is adversely affected for some applications.
- 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) the average overhead is 89%, but with equal or better speedup than running all modules speculatively for all the benchmark applications.

Overall, this study shows that it is possible to exploit most of the inherent speculative module-level parallelism and still remove most of the overhead associated with thread management.

Acknowledgments

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

References

[1] 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, February 2002.

[2] 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, October 1999.

[3] 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, February 1998.

[4] 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, October 1998.

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

[6] 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.

[7] 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.

[8] 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)*. IEEE, May 1997.

[9] K. Olukotun, L. Hammond, and M. Willey. Improving the performance of speculatively parallel applications on the hydra CMP. In *Proceedings of the 1999 International Conference on Supercomputing (ICS '99)*, pages 21–30. ACM Press, June 1999.

[10] 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, June 2001.

[11] 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.

[12] 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, November 1997.

[13] 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)*. IEEE Computer Society, February 2002.

[14] 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, February 1998.

[15] 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, October 1996.

[16] 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, September 2001.

[17] 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), to appear*. IEEE Computer Society, April 2003.