McGill University
School of Computer Science
Sable Research Group

Asymmetric Partitioning in Thread-Level Speculation

Sable Technical Report No. sable-2015-2

Alexander Krolik, Clark Verbrugge

November 2015

Contents

List of Figures

Abstract

Software-based Thread-Level Speculation (TLS) requires speculative threads executing ahead of normal execution be isolated from main memory until validation. The resulting read/write buffering requirements, however, mean speculative execution proceeds slower than unbuffered, non-speculative execution, resulting in poor parallel coverage of loops as the non-speculative threads catches up to and prematurely joins with its speculative children. In this work we investigate an "asymmetric partitioning" strategy, modifying speculative code generation to better partition loops, balancing the load assuming a range of constant factor slowdowns on speculative threads. An implementation within the LLVM-based MUTLS system shows a significant benefit to memory intensive benchmarks is possible, although it is dependent on relatively precise estimates of the memory access rate that induces buffering slowdown for individual benchmarks.

1 Introduction

Thread-Level Speculation (TLS) is a safe technique to automatically parallelize sequential programs; employing otherwise idle cores to speculatively execute sections of a program ahead of time. This execution is not without cost. The main sequential execution is responsible for starting and stopping speculative execution, while speculation must run in an isolated container from the main program and therefore results in slower memory access.

In the context of loops, Loop-Level Speculation (LLS) distributes loop iterations over available CPUs thus running multiple iterations of a loop at the same time. Due to speculation costs of repeatedly starting and stopping speculative execution, it is not always practical to run each iteration on a separate thread. The technique of *loop partitioning* attempts to reduce this cost by running multiple iterations on each thread. This does not factor in the cost of running in an isolated state, so standard loop partitioning results in less parallelization than optimal. Asymmetric partitioning attempts to optimally distribute loop iterations over the non-speculative and speculative threads therefore achieving better parallel coverage. This in turn can lead to improved program performance.

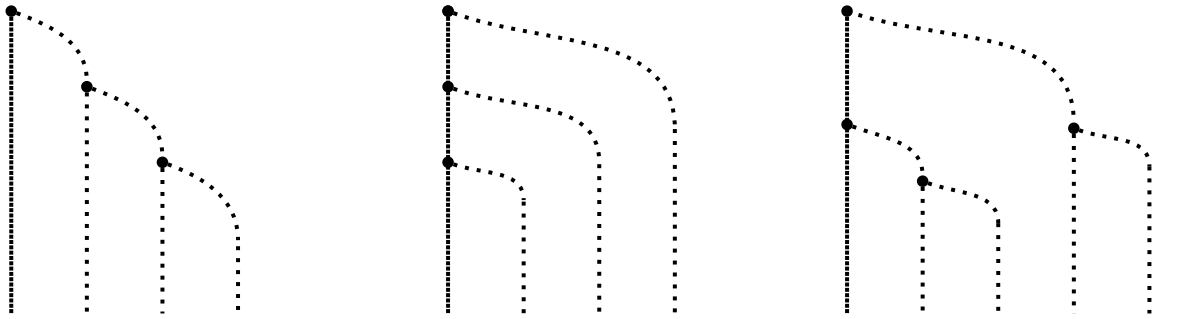
An existing software TLS implementation, MUTLS, allows implementation of this technique at compile time [1]. This is accomplished through splitting non-speculatuve iterations from speculative iterations, then symmetrically partitioning the speculative iterations to run on multiple speculative threads. Results indicate that speculative execution that experiences slowdown due to memory access can benefit from asymmetric partitioning, while CPU intensive benchmarks which do not experience these slowdowns show no improvement.

2 Background

Thread-Level Speculation (TLS) is a safe technique to automatically parallelize sequential programs; employing otherwise idle cores to speculatively execute sections of a program ahead of time. This is achieved through successive pairs of *fork*, *join* and optionally *barrier* points that indicate locations to begin and commit speculative work. When a program encounters a *fork* point, sequential execution continues on a *non-speculative* parent thread, while a *speculative* child thread is launched at the paired *join* point. To guarantee safety of speculative execution, all reads and writes on a

speculative thread are buffered. Upon reaching the join point, the non-speculative parent thread joins with its speculative child threads by halting and validating their execution. Validation ensures that the necessary parts of the environment used for speculative execution are the same as what would have been used in a normal sequential execution. Successful validation commits the execution of the speculative threads to memory and continues execution from the termination point of the speculative thread. Unsuccessful validation discards the speculative execution and re-executes the speculative work on the parent thread. In addition to a parent thread signaling it wishes to join, a child speculative thread may also stop executing when it encounters a *barrier* point. Barriers are indications that it may not be safe or beneficial to speculatively execute past this point. The end result of successful speculation is a safety guaranteed parallel execution of an intially sequential program.

To achieve greater speedup and maximize the use of cores on a multi-core machine, multiple speculative threads can be launched at various points in a program. There are 3 main models for multiple-speculation that define the speculative thread tree: *in-order nesting*, *out-of-order nesting*, and *mixed nesting* as shown in Figures 1a, 1b and 1c respectively.



(a) *in-order*: Only the most speculative thread may fork

(b) *out-of-order*: Only the least speculative thread (non-speculative) may fork

(c) *mixed*: Both *in-order* and *out-of-order*, any thread may fork

Figure 1: Forking models

While mixed speculation provides the best overall solution since it includes all combinations of in-order and out-of-order thread hierarchies, both in-order and out-of-order speculation provide optimal solutions to some problems [6]. Thus, by using the effective multiple speculation model and forkpoints, greater speedup and more parallelism can be obtained.

While speculative execution may improve performance through parallelism, there are runtime costs which affect optimal performance. Both forking and joining speculative threads must perform several housekeeping operations to maintain an accurate representation of the speculative system. In addition, joining a speculative thread requires validating the execution by comparing the expected inputs versus the actual inputs. There is also a speculative execution penalty. During non-speculative execution, threads read and write directly to and from memory. In order to guarantee safety, speculative threads must read and write to a buffer that models the environment of the expected non-speculative execution. This buffering comes at a cost:

Non-speculative execution time: sequential non-spec execution time + fork cost + join cost

Speculative execution time: γ(non-spec execution time)

where γ denotes the slowdown function due to speculation. Therefore, the costs associated with speculation affect the end performance of the program.

In addition to flexible TLS implementations that handle arbitrary forking and joining, there exist more limited models which target specific program sections. Loop-Level Speculation (LLS) applies the principles of TLS in-order speculation to speculate on the loop body for future iterations. The parent, non-speculative thread executes the first iteration, and effectively spawns an additional k speculative child threads to execute the following k iterations of the loop. In this model, 1 thread executes 1 iteration. Upon joining with its child threads (assuming successful validation), the parent thread then executes the $(k+2)$-th iteration and can optionally spawn more child threads to continue speculative execution. As a limiting case, for a loop of size n, a total of $n-1$ speculative threads could be launched in-order to execute all n iterations of the loop in parallel. Therefore we can apply the principles of TLS to a specific programming construct to obtain effective speedup.

Due to speculation costs, a straightforward LLS implementation is not effective. With small loop bodies, the cost of repetitive forks and joins outweights the benefit of using multiple threads. Instead, a technique called *loop partitioning* can be used. This technique divides a loop of size n into N *blocks*, each of which execute $\frac{n}{N}$ iterations. The result is a non-speculative thread which executes $\frac{n}{N}$ iterations, and $N-1$ speculative threads which each execute $\frac{n}{N}$ iterations. The initial value and end value of each block is determined by the following relationship

$$s_j = \frac{n}{N} \cdot j$$

$$t_j = \frac{n}{N} \cdot (j+1) - 1$$

The code transformation is shown in Figure 2, where each iteration of the outer loop executes a single block - the nested loop. Since forking and joining is only once per block, this is less costly than a standard LLS implementation. Thus loop partitioning is an effective method to parallelize loops and minimize speculation costs.

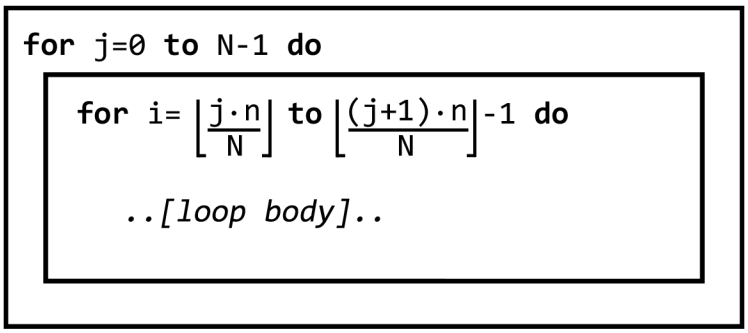


Figure 2: Symmetric partitioning

2.1 MUTLS

The MUTLS (Mixed Model Universal Software Thread-Level Speculation) project is a software TLS implementation in LLVM (Low Level Virtual Machine) that uses automatic or user defined fork/joinpoints to choose speculation locations [2]. LLVM is a compiler framework which allows compiler writers to write compilers that accept a variety of input languages, apply a set of optimizations and output to a supported machine language. To do this code generation, an SSA Intermediary Representation (IR) is used. MUTLS uses the mixed model of speculation to achieve maximum levels of parallelism. In addition to automatically marking fork and join points, users of the project can also mark their own forkpoints, joinpoints, barrierpoints, partitioned loops for use in the speculative system by using the pragmas illustrated in Figure 3.

```
#pragma tls forkpoint [id <id>]
#pragma tls joinpoint [id <id>]
#pragma tls barrierpoint [id <id>]
#pragma tls forkpoint loopblock [id <id>]
```

Figure 3: MUTLS forkpoint marking

3 Method

While loop partitioning addresses the overhead of forking and joining for each iteration, it does not efficiently distribute the workload over the CPU resources. Traditionally, *symmetric* loop partitioning consists of dividing a loop of n iterations into N blocks, each with an equal number of iterations. The advantage of this approach is the reduced speculation cost over repeatedly forking and joining for each iteration. This addresses the overhead of forking and joining, however it does not take into account the cost of buffered reads and writes that are included in the γ slowdown function. The result of this approach is speculative threads which want to run for longer than their non-speculative parent thread. In an ideal distribution, to minimize the runtime of the program, speculative threads should terminate at the same time as their parent threads. This ensures maximum parallelization as no thread is waiting for another thread to terminate, and there are no early joins that result in sequential execution.

In order to achieve higher levels of parallel execution, the workload between speculative blocks and the non-speculative parent block is split in an *asymmetric* fashion. Since non-speculative blocks execute faster than their speculative child blocks, more iterations can be allocated to non-speculative execution to balance the runtime of different threads. This *bias factor*, the ratio of work done by a non-speculative block and work done by a speculative block, is closely tied to the slowdown introduced by the read and write buffering. A higher bias indicates a larger amount of work being executed non-speculatively while speculative blocks execute less. The optimal asymmetric distribution of loop iterations allows for maximum parallelization and can therefore increase the performance of the program.

Asymmetric partitioning is done in 3 main steps. Firstly, in order to allow for an asymmetric distribution of iterations between speculative and non-speculative threads, the initial loop is split

into two identical copies. The two copies are then rewired to start and stop according to a calculated split iteration which indicates the split between non-speculative and speculative iterations. In the second phase, the speculative loop is symmetrically partitioned according to the remaining blocks to distribute the speculative work over multiple speculative threads. Finally, to control the multithreaded execution fork, join and barrier points are inserted into the asymmetrically partitioned loops.

3.1 Loop Splitting

In order for non-speculative and speculative threads to run a different number of iterations and for multiple speculative threads to be allowed, the initial loop must first be split into two identical copies. The end result of loop splitting is to split the loop into two copies, one which executes the non-speculative iterations and one which executes the speculative iterations. Since the asymmetric block counts towards the total blocks count, it must obey the same threshold as a symmetric block. The threshold check is calculated as follows for a bias factor k, N blocks and n iterations:

$$\frac{n \cdot k}{N + k - 1} \geq \text{minimum block size}$$

If the threshold check is passed, then the loop splitting can continue. Since not all loop counts are known during the compilation phase, the threshold check is only used when a constant number of iterations is available.

Another important consequence of a non-constant loop count is when the bias allocates zero iterations to the non-speculative thread. During development this caused the creation of infinite loops, since the exit condition was built with the assumption of some iterations occuring. To fix this issue, a *skip check* is inserted before the non-speculative loop. Should the skip check conclude there are no iterations in the non-speculative loop, it will jump to the speculative loop. In this case, the number of blocks is decreased by one, and the first iteration of the symmetric partition becomes the non-speculative execution. Should loop splitting proceed, the result can be seen in Figure 4. The following section explains the bounds.

```
for i=0 to ⌊k·n / N+k-1⌋-1 do
  ..[loop body]..
```

```
for i=⌊k·n / N+k-1⌋ to n-1 do
  ..[loop body]..
```

Figure 4: Loop splitting

3.2 Asymmetric Bias

The goal of assymmetric partitioning is for the non-speculative thread to execute a factor k times the amount of work of a speculative thread. For asymmetric bias factor k, N blocks, and n iterations, the allocation of loop iterations is as follows:

- Fraction of blocks allocated to non-speculative loop: $\frac{1}{N}$
- Fraction of blocks allocated to speculative loop: $\frac{N-1}{N}$
- Fraction of iterations allocated to non-speculative loop: $\frac{k}{N+k-1}$
- Fraction of iterations allocated to speculative loop: $\frac{(N-1)\cdot 1}{N+k-1}$
  where each block receives $\frac{1}{N+k-1}$.
- Iterations allocated to non-speculative loop: $\frac{k}{N+k-1} \cdot n$
- Iterations allocated to speculative loop: $\frac{N-1}{N+k-1} \cdot n$
  where each block receives $\frac{n}{N+k-1}$ iterations.

As an example, take $k = 2, N = 5, n = 90$

- Fraction of blocks allocated to non-speculative loop: $\frac{1}{5}$
- Fraction of blocks allocated to speculative loop: $\frac{4}{5}$
- Fraction of iterations allocated to non-speculative loop: $\frac{2}{5+2-1} = \frac{1}{3}$
- Fraction of iterations allocated to speculative loop: $\frac{(5-1)\cdot 1}{5+2-1} = \frac{2}{3}$
  where each block receives $\frac{1}{6}$.
- Iterations allocated to non-speculative loop: $\frac{1}{3} \cdot 90 = 30$
- Iterations allocated to speculative loop: $\frac{2}{3} \cdot 90 = 60$
  where each block receives 15 iterations.

In addition to supporting integers, this division of iterations also holds for decimal quantities. To avoid premature rounding with the LLVM *Scalar Evolution* classes which perform integer division, the following approximation is used to allow an arbitrary but finite precision. Let $k_f = \frac{k_n}{k_d}$ be a fractional approximation of bias factor k.

$$\begin{aligned}
\frac{n \cdot k}{N + k - 1} &\approx \frac{n \cdot k_f}{N + k_f - 1} \\
&= \frac{n \cdot \frac{k_n}{k_d}}{N + \frac{k_n}{k_d} - 1} \\
&= \frac{n \cdot k_n}{N \cdot k_d + k_n - k_d}
\end{aligned}$$

We can then define k_f to be a fraction representing bias k with arbitrary precision. In this particular implementation, 2 decimal places of precision were allowed, so

$$k \approx k_f = \left\lfloor \frac{k \cdot 100}{100} \right\rfloor$$

In general, for precision p

$$k_f = \left\lfloor \frac{k \cdot 10^p}{10^p} \right\rfloor$$

3.3 Partitioning

In order to have multiple speculative threads, the cloned loop is partitioned. A symmetric partioning is used, since all threads running the cloned iterations are designed to run speculatively and thus run at the same speed. The number of blocks for the symmetric partitioning depends on the compile-time success of loop splitting. If the loop is successfully split, then the partitioning number will be $N-1$. Should the loop fail to split at compile-time, due to threshold conditions or a skip check, then the full N blocks will be allocated to the symmetric partitioning. The full asymmetric partitioning result can be seen in Figure 5.

```
for i=0 to ⌊(k·n)/(N+k-1)⌋-1 do
  ..[loop body]..
```

```
for j=0 to N-2 do
 let s=⌊(k·n)/(N+k-1)⌋
 let n'=n-s
 for i=⌊(j·n')/(N-1)⌋+s to ⌊((j+1)·n')/(N-1)⌋+s-1 do
    ..[loop body]..
```

Figure 5: Asymmetric partitioning

3.4 Fork Points

In contrast to a symmetric partitioning, in an asymmetric partitioning there are multiple fork and join points inserted into the code. The final goal is to have a single non-speculative thread executing the non-speculative loop, and $N-1$ threads executing the $N-1$ speculative blocks from the symmetric partitioning (assuming successful split). To do so, a fork must be performed before the start of the non-speculative loop, but after the skip check is performed. This placement allows the non-speculative loop to be run on a non-speculative thread if possible. If not, then the non-speculative thread will execute the first block of the symmetric loop. The paired join point is placed after the termination of the loop so the speculative thread start executing the speculative loop. Note that in order to prevent waiting for a thread which was never forked, the join point must only be run if the skip check returned false. We thus have 2 basic blocks which are predecessors to the speculative loop.

The symmetric fork and join points are inserted as per a normal symmetric partitioning. That is, fork before the inner loop, and join directly after. This effectively spawns one speculative thread per outer loop iteration, and thus one speculative thread per block. In order to stop execution once the inner loop is complete and prevent unnecessary rollbacks, a barrier point is inserted directly after the join point. We then have the final picture shown in Figure 6.

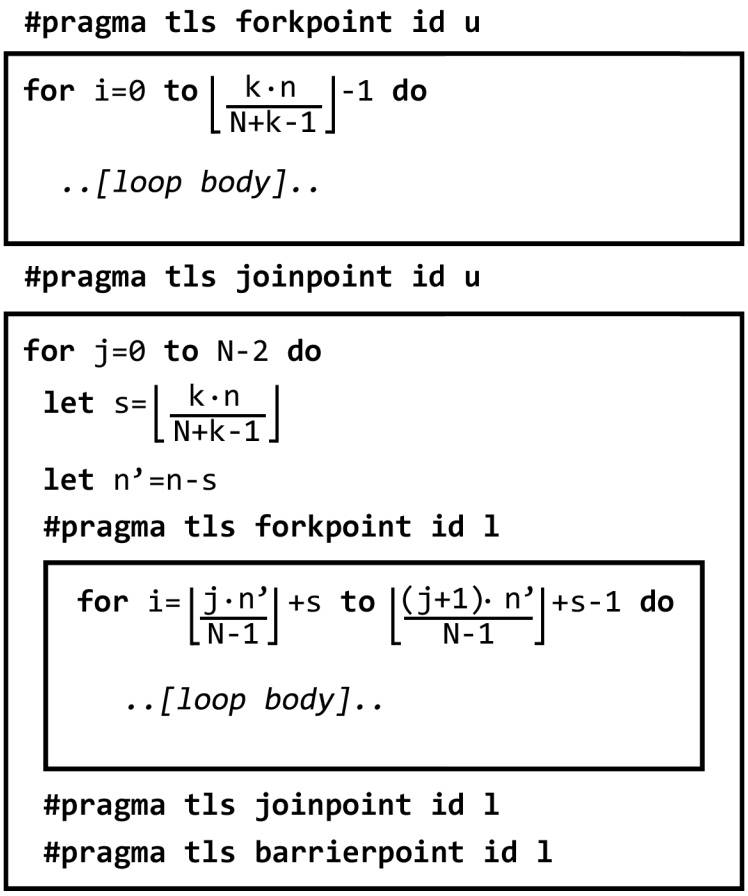


Figure 6: Asymmetric partitioning with fork and join points

4 Results

Results were collected on a 4 x Intel(R) Xeon(R) CPU E7-4850 @ 2.00GHz server, 10 cores hyper-threaded, 24MB L2 cache, and 64 GB memory.

Name	Description	Problem Size	Intensive
synthetic	simulated buffering on speculative threads with 2x slowdown	500 loop iterations, $W = 10^8$	CPU
md	3D molecular dynamics	256 particles, 400 step	CPU
3x+1	3x+1 number theory problem	$N = 1280000$	CPU
barneshut	N-body simulation	$N = 12800$ bodies	Memory
loopmatmult	block based matrix multiplier	1024×1024 matrices	Memory
mandelbrot	factal generation	512×512 image 80000 iterations maximum	CPU

The legend shown in Figure 7 is used for the results graphs. Note that due to consistency in

results, the standard deviation is not always visible. Blocknum indicates the number of threads and thus the number of blocks used for the benchmark.

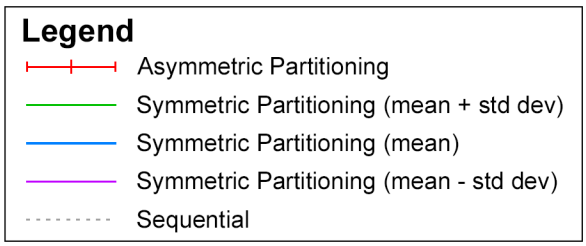


Figure 7: Results legend

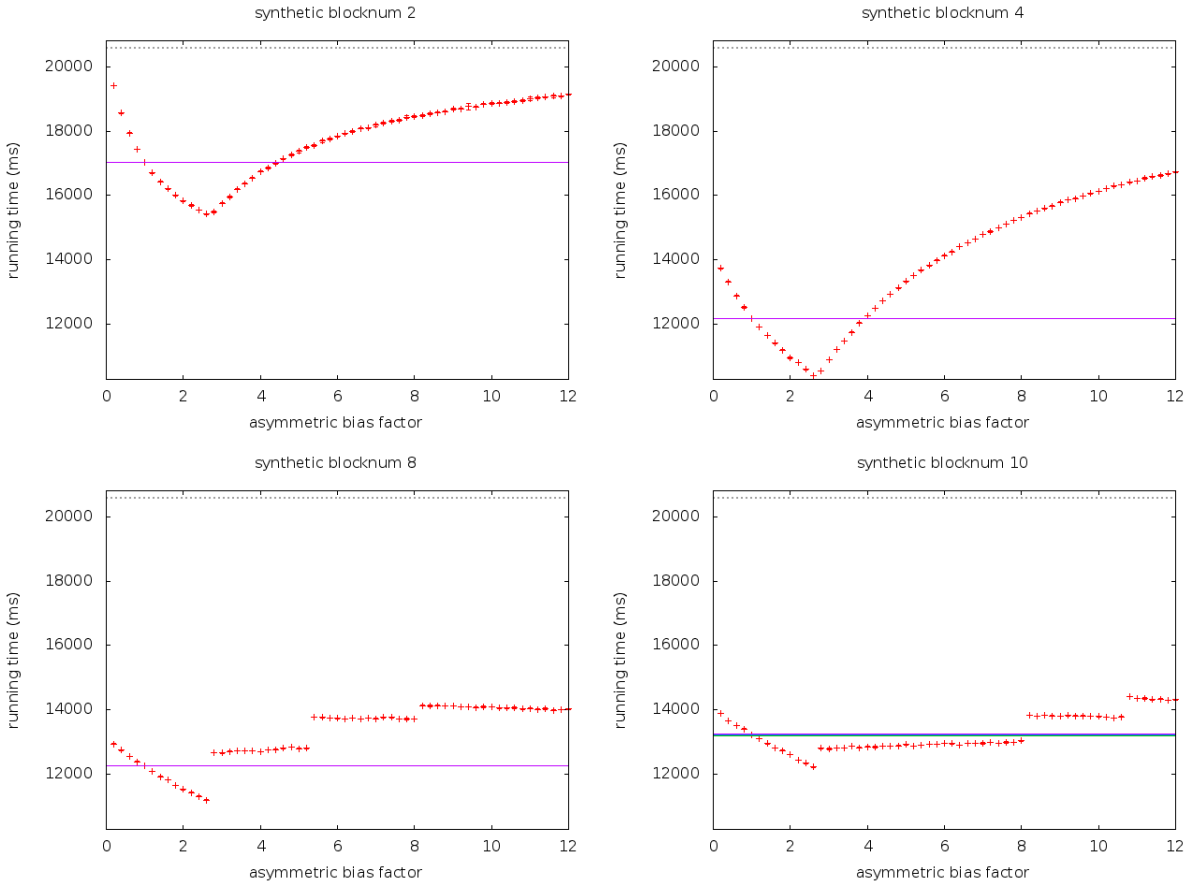


Figure 8: Synthetic buffering benchmark, run at 2, 4, 8, and 10 blocks, optimal at 2.5

As a method of exploring the relationship between the slowdown and the asymmetric bias factor, a synthetic test was designed to simulate speculative slowdown. Before executing its work, a loop iteration checks to see if it is being executed speculatively or non-speculatively. In the cast of

non-speculative execution, W units of work are performed. If the check determined the work was executing speculatively, $2 \cdot W$ units of work are performed, simulating a slowdown factor of 2x. As shown in Figure 8, a local optimal bias of 2.4-2.6 is present in all cases, suggesting a tight relationship between speculative slowdown and the optimal bias factor. This observation is also supported by the benchmark set, with CPU intensive benchmarks having lower optimal bias factors than memory intensive ones.

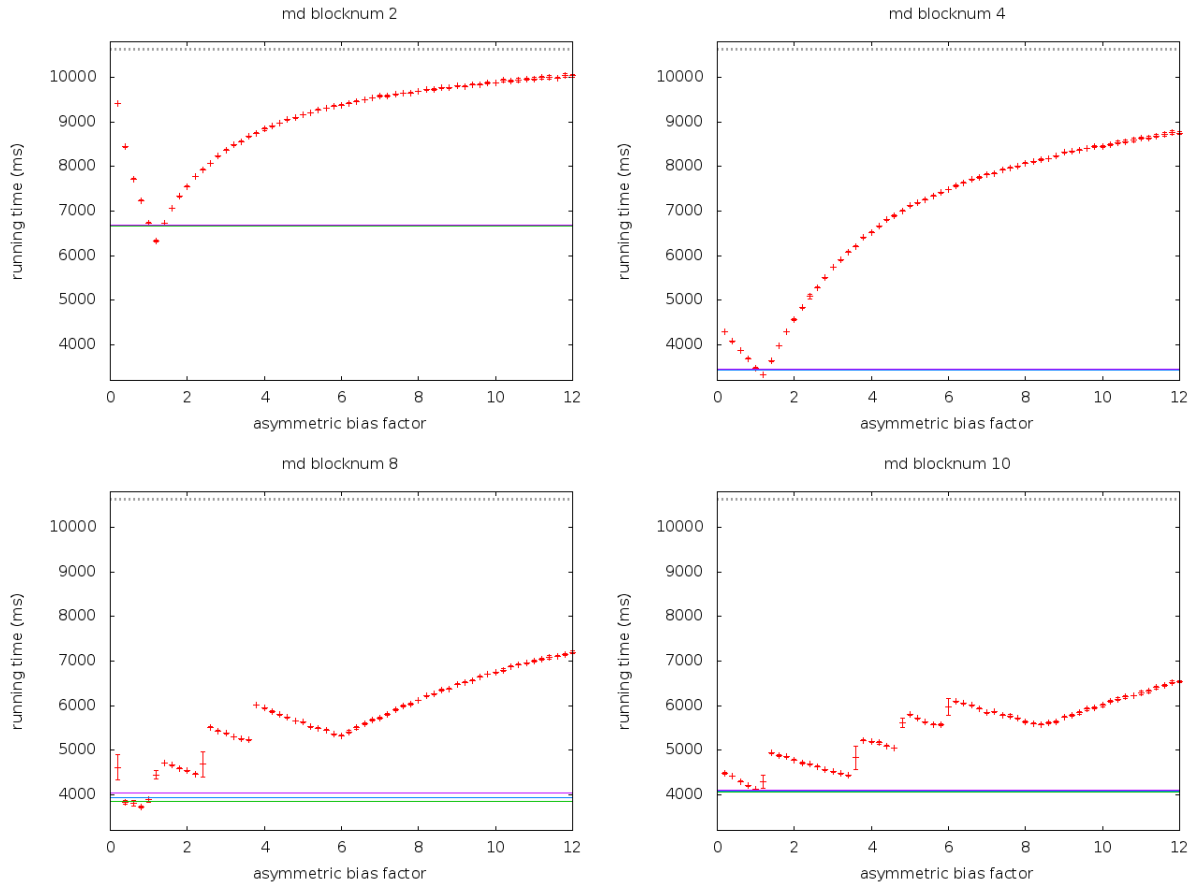


Figure 9: moldyn 3D molecular dynamics, run at 2, 4, 8, and 10 blocks, optimal at 1

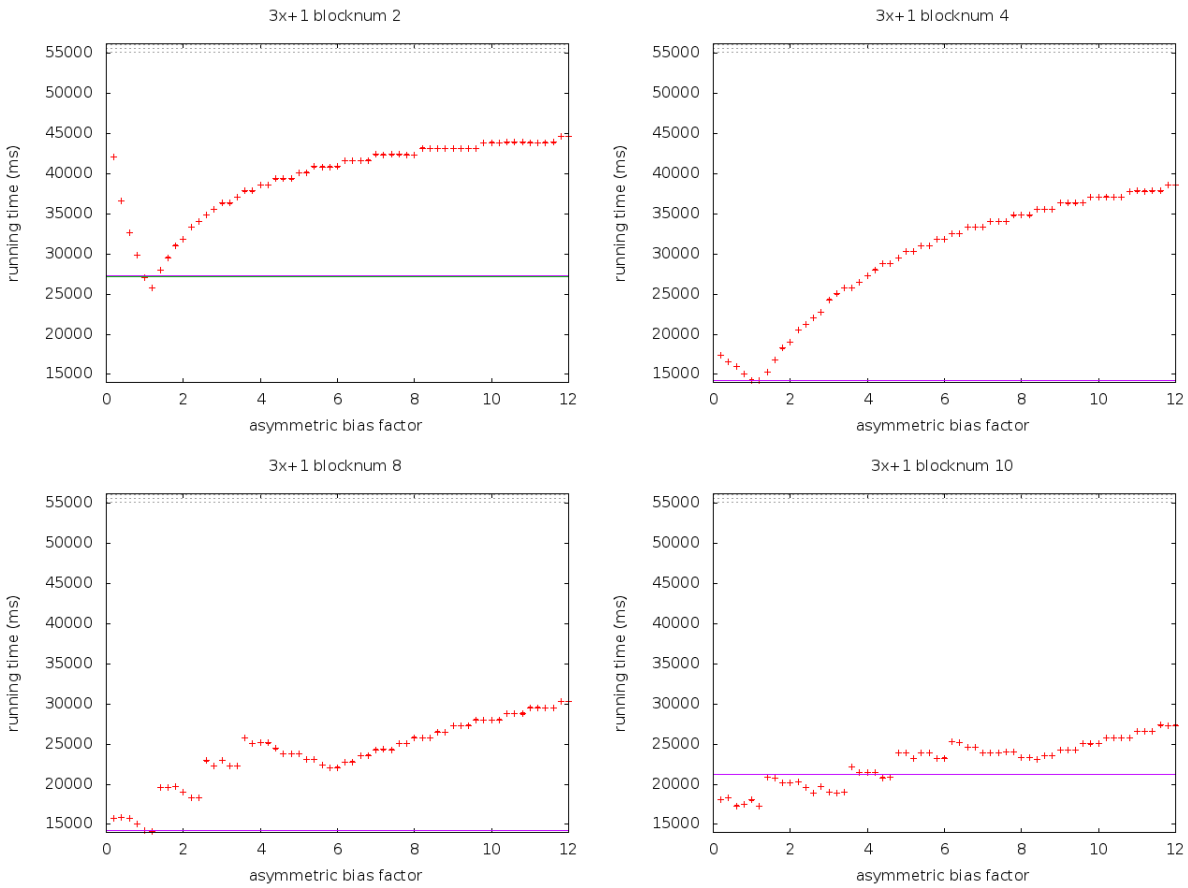


Figure 10: 3x+1 number theory benchmark, run at 2, 4, 8, and 10 blocks, optimal at 1

In general, CPU intensive benchmarks do not have the same slowdown as memory intensive benchmarks during speculative execution. This is largely due to the minimal effect buffering has on CPU operations. Thus the non-speculative and speculative threads of a CPU intensive benchmark will run at approximately the same speed, an optimal iteration distribution will allocate the same number of iterations to speculative and non-speculative blocks. As seen in Figures 9 and 10, a symmetric partitioning performs consistently better than any asymmetric bias. Note that a bias of around 1.0 will produce a similar result to a symmetric partitioning. This is due to the work being evenly distributed to both non-speculative and speculative threads, so at runtime this emulates a symmetric partitioning with the same number of blocks.

Load balance plays an important part in the results, especially at higher block numbers. Benchmarks start exibiting a *saw tooth* type behaviour, with local optimal biases followed by a loss in performance. This behaviour is extremely evident with CPU intensive benchmarks such as moldyn and 3x+1 as seen in Figures 9 and 10 with 8 and 10 threads. Around the jump locations, there is also a higher degree of variance, with both optimal and low performance results obtained. An analysis of the individual results showed that at the jump points either a local minimum or a local maximum was obtained, but never in between. This type of behaviour is believed to be from a load imbalance when the non-speculative thread signals to join. In some cases the threads complete at

the same time, and in others additional work is required once the join is complete or one thread waits for the completion of another.

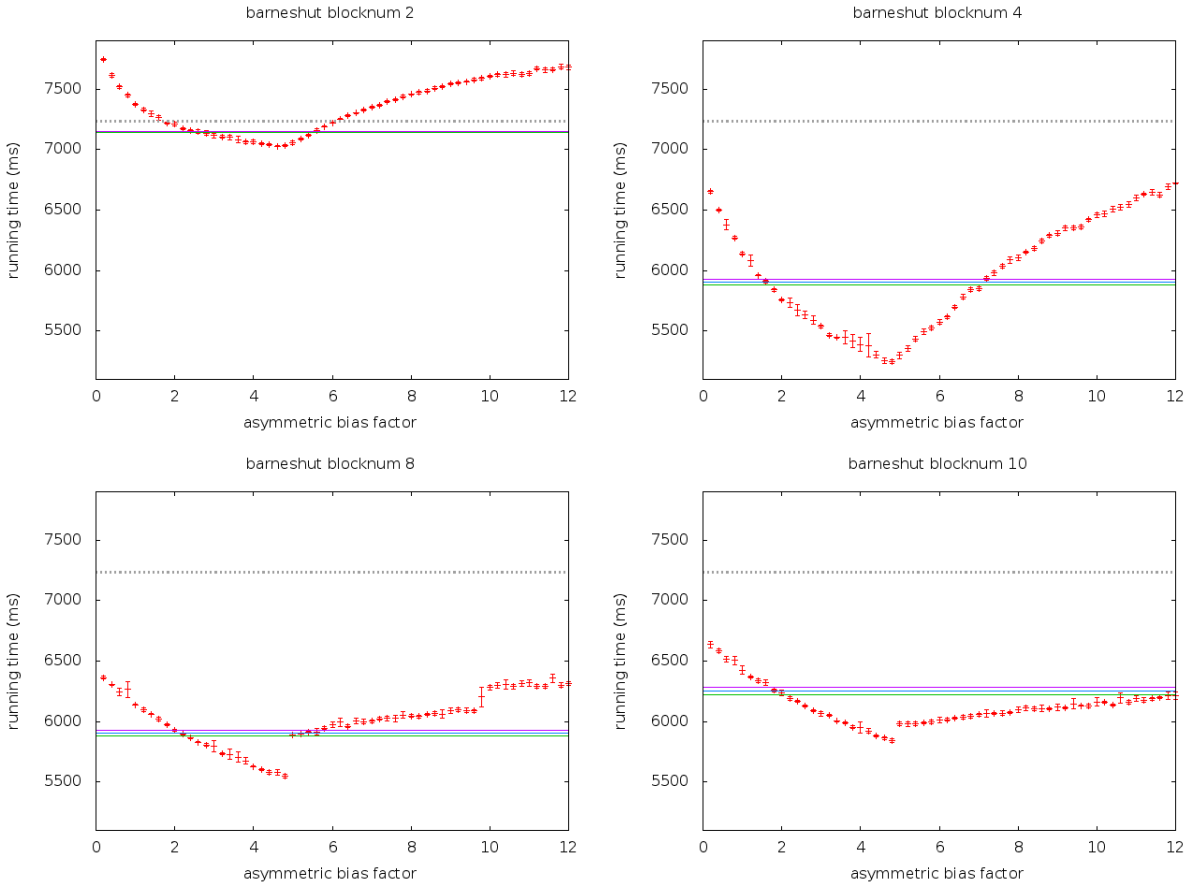


Figure 11: barneshut N-body simulation, run at 2, 4, 8, and 10 blocks, optimal at 4.5

Memory intensive benchmarks however can have significant improvement at the optimal bias factor. Due to slowdowns of speculative threads, a symmetric partitioning does not provide optimal parallelization. This results in the non-speculative thread accumulating work from speculative threads and thus taking longer to execute. By providing the optimal bias factor, we optimize parallel coverage, and thus the non-speculative thread completes at the same time as its speculative child threads.

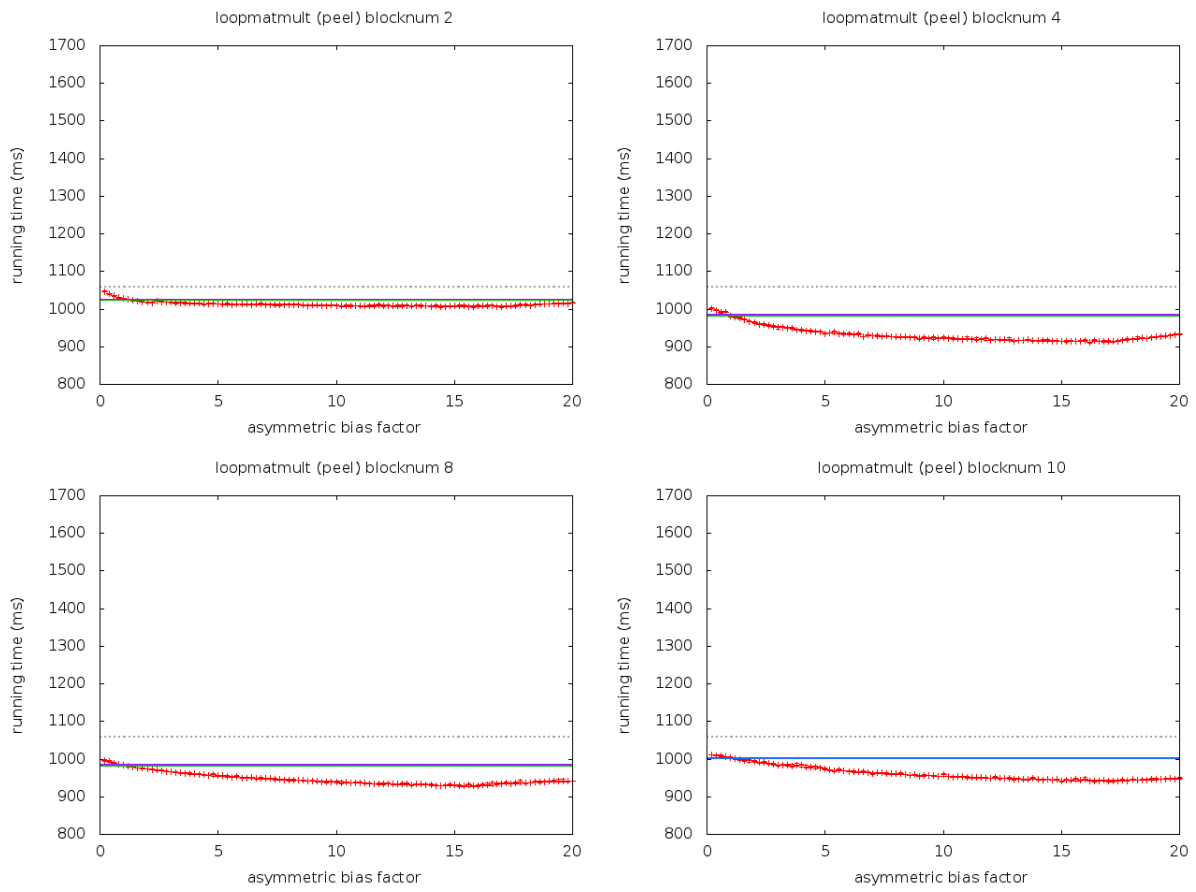


Figure 12: loop matrix multiplier, run at 2, 4, 8, and 10 blocks, optimal at 15

As shown in Figure 12, the loop matrix multiplier benchmark showed no improvement at all with any bias factor, so loop peeling was performed with 1 iteration to improve caching performance. This resulted in an optimal bias that improved performance over both sequential and symmetric partitioning. The symmetric partitioning performance was also observed to be constant no matter which block number was chosen. This could be due to a limiting performance improvement that is inherent to the benchmark. Asymmetric partitioning also showed the same behaviour with little to no improvement from increasing the number of threads.

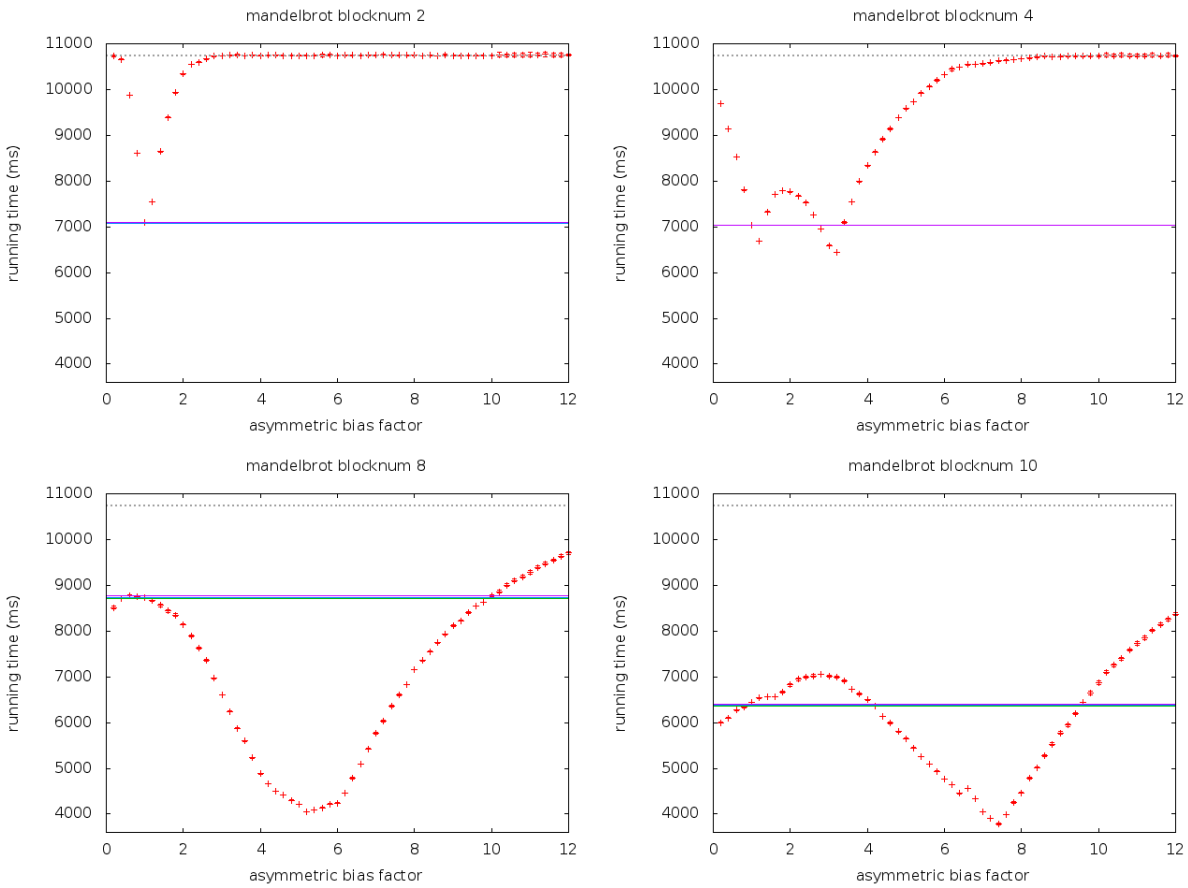


Figure 13: mandelbrot fractal generation, run at 2, 4, 8, and 10 blocks, optimal ¿ 1

Asymmetric performance was also limited in some cases by the sequential execution. In the mandelbrot benchmark, when a high enough bias factor was chosen, therefore allocating a very high percentage of work to non-speculative execution, the result was limited by the sequential execution as shown in Figure 13. At these high biases, no work is executed speculatively, so the end result is a non-speculative thread performing all iterations. In general, after passing the optimal bias, asymmetric performance levels off around sequential execution. The exception to this case is barneshut, which experiences some slowdown over sequential execution. This could be due to caching effects, fork and join costs, or a load imbalance.

5 Related Work

There have been several attempts to improve performance and parallel coverage of Loop-Level Speculation. Wang et al. [7] build a DAG loop graph of the program to model loop nesting relationships. Using coverage and speculation speedup estimates, their algorithm chooses a set of loops for Loop-Level Speculation to optimize performance.

Llanos et al. [3] proposed the use of Just-In-Time scheduling for dynamically computing the block

size for block-based Loop-Level Speculation. Their scheduling algorithm improves performance of LLS in loops with dependencies by 10-26% and loops without dependencies by 9-39%. In addition, by including dependency violation information in the block size calculation, the number of violations is reduced. For loops with frequent inter-iteration depencencies this reduces the misspeculation cost incurred by using a fixed block size.

Samadi et al. [5] explored load balancing LLS between CPUs and GPUs. The system introduced, *Paragon*, executes loops speculatively on GPUs and uses a lightweight management system to monitor and solve dependencies. Since GPUs and CPUs run concurrently, the rollback cost is minimized. On average, *Paragon* achieves a 12x speedup compared to an unsafe 4 thread CPU code.

Oplinger et al. [4] examined performance of Loop-Level Speculation with value prediction in the context of realistic programs. When only the best loop in a nesting relationship was chosen, performance increased by a harmonic mean of 1.6. With multi-level loop speculation, performance increased to 2.6. However, their study found that speedup was improved by also performing procedural speculation.

6 Conclusion

Asymmetric partitioning provides a foundation for load balancing loop iterations between non-speculative and speculative threads. While CPU intensive benchmarks do not show any significant improvement due to limited buffering, memory intensive programs can benefit immensely from the optimal asymmetric bias. This points to a direct relationship between speculative slowdown and the optimal bias factor. The more buffering that needs to be done, the higher the bias factor required in order to maximize parallel coverage.

As future work, heuristics could be used to determine the optimal bias factor, depending on how much buffering would be done by the speculative system. In addition, choosing the right block number for the system could provide maximum program performance. This could be done at compile time with knowledge of the machine, but dynamic recompilation or runtime checks could be used to choose the appropriate bias and block numbers.

References

[1] Z. Cao. Mutls: Mixed model universal software thread-level speculation. `http://www.sable.mcgill.ca/~zcao7/mutls/`, Sept. 2013.

[2] Z. Cao and C. Verbrugge. Mixed model universal software thread-level speculation. In *ICPP: International Conference on Parallel Processing The 42nd Annual Conference*, pages 651–660. IEEE, Oct 2013.

[3] D. Llanos, D. Orden, and B. Palop. Just-in-time scheduling for loop-based speculative parallelization. In *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on*, pages 334–342, Feb 2008.

[4] 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–, Washington, DC, USA, 1999. IEEE Computer Society.

[5] M. Samadi, A. Hormati, J. Lee, and S. Mahlke. Paragon: Collaborative speculative loop execution on gpu and cpu. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 64–73, New York, NY, USA, 2012. ACM.

[6] C. Verbrugge, A. Kielstra, and C. J. Pickett. Abstract analysis of method-level speculation. Technical Report SABLE-TR-2011-3, Sable Research Group, School of Computer Science, McGill University, Montréal, Québec, Canada, 2011.

[7] S. Wang, X. Dai, K. Yellajyosula, A. Zhai, and P.-C. Yew. Loop selection for thread-level speculation. In E. Ayguad, G. Baumgartner, J. Ramanujam, and P. Sadayappan, editors, *Languages and Compilers for Parallel Computing*, volume 4339 of *Lecture Notes in Computer Science*, pages 289–303. Springer Berlin Heidelberg, 2006.