Speculative Precomputation

Carlos Augusto S Cunha

Departmento de Informática
Universidade do Minho
cacunha@di.estv.ipv.pt

Abstract. *Current processors are based on a multithreaded architecture. Simultaneous Multithreading (SMT) techniques are used to increase instruction throughput under a multiprogramming workload; however, it does not improve performance when only a single thread is executing. This communication explores Speculative Precomputation, a technique that uses idle thread contexts in a multithreaded architecture to improve the performance of single-threaded applications. It reduces program stalls, generated by data cache misses, using available thread contexts to prefetch these data. Speculative Precomputation tasks are described, and the examination of basic triggers and chaining triggers are made, using a Intel Itanium ISA processor as the evaluation base.*

1 Introduction

The difference between the speed of computation and the speed of memory access continues to grow (CPU-memory gap). On the other hand, the actual processors have Simultaneous Multithreading (SMT) characteristics, improving overall throughput under a multiprogramming workload. Consequently, the use of various thread contexts on SMT, only improve the system performance when there are more than one thread executing.

The Speculative Precomputation (SP) is a technique to improve single-thread performance on a multi-threaded architecture.

The basic idea of SP is to use idle hardware thread contexts to execute speculative threads that attempt to avoid future cache misses, loading the values in advance from memory and passing them to non-speculative thread. Speculative precomputation can be thought of as a special prefetching mechanism that can be used when data does not exhibit predictable access patterns, so it is not possible to prefetch via traditional mechanisms.

SP targets static loads that cause the most stalls in the non-speculative thread. In most programs the set of delinquent loads is quite small; commonly 10 or fewer loads cause more than 80% of L1 cache misses [1]. In Figure 1, we can see the cumulative contribution of the worst behaving static loads to L1 data cache misses. It is evident that a few poorly behaved static loads dominate cache misses in these programs. We call these loads delinquent loads.

The delinquent loads extracted from the non-speculative thread, are grouped in precomputation slices (p-slices), and executed in advance to precompute the address that will be accessed by a future delinquent load, and prefetches the data.

For the basis of performance gains comparison, a research SMT processor will be used, implementing the Intel Itanium processor instruction set.

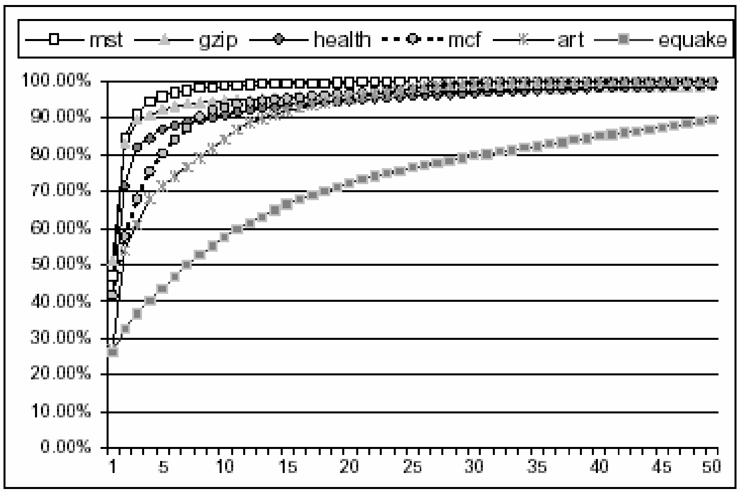


Figure 1. Cumulative L1 data cache misses contributed by the worst behaving static loads (Courtesy of Jamison D. Collins [2])

The remainder of the paper is organized as follows. Section 2 discusses related research. Section 3 presents some multithreading basic aspects related with Itanium architecture. Section 4 explains all techniques used by Speculative Precomputation to identify delinquent loads and to spawn and manage p-slices. Section 5 analyses the Software based SP approach (SSP) and its implementation on the Itanium processor. Section 6 covers chaining triggers. Section 7 concludes.

2 Related work

SP works by identifying and extracting the small number of static loads, known as delinquent loads, that are responsible for the vast majority of memory stall cycles. Its mandatory to analyse the code to find static loads. That work can be made with user intervention or not. The basic steps and algorithmic ingredients for Speculative Precomputation can be implemented using many techniques, ranging from a hardware-only approach, to a software-only approach.

Several thread-based prefetching paradigms based on SP have been proposed previously. Collins et al.'s Speculative Precomputation [2] explores the software-based SP (SSP) on Itanium processor with very little hardware support to implement SP.

Hong Wang et al.'s Exploring the use of Multithreading for latency [3], explores the combination of SSP and Out of Order execution techniques to speed up a set of pointer-intensive applications on a version of Intel Xeon processors with Hyper-Threading Technology.

Collins et al.'s Dynamic Speculative Precomputation [4] proposes a hardware approach, using back-end instruction analysis hardware, located off the processor's critical path, to perform all necessary runtime instruction analysis, extraction, and optimization without user intervention.

Luk's Software controlled Pre-Executions [5] focuses on the use of specialized, compiler inserted code that is executed in available thread contexts to provide prefetches for a non-speculative thread.

This work focus on an overall approach to Speculative Precomputation, passing by a realistic SP implementation using an Intel Itanium processor.

3 Multithreading

We assume in this research that a single non-speculative thread occupies one thread context while the remaining thread contexts are either idle or used by speculative threads. Each hardware thread context of the modelled Itanium processor has a private, per-thread expansion queue and register files. All visible registers, including 128 general purpose registers (GR), 128 fp registers (FR), 64 predicate registers (PR) and 128 control registers are replicated for each thread.

If more than one thread is ready to fetch or execute, two threads are selected from those that are ready, and each is given half of the resource bandwidth. A round-robin policy is used to prioritize the sharing between threads.

4 Speculative Precomputation

In speculative precomputation, a non-speculative thread (main thread) spawns a p-slice on a new thread context that will compute and prefetch an address expected to be accessed by a delinquent load in the near future. To construct the p-slices it is necessary to identify the delinquent loads that are responsible for the majority of cache misses.

4.1 *SP tasks*

Speculative Precomputation follows a set of mandatory steps: identification of delinquent loads, construction of p-slices for these loads, and the establishment of triggers. This work can be performed with compiler assistance [2][3] as well as some hardware support [4].

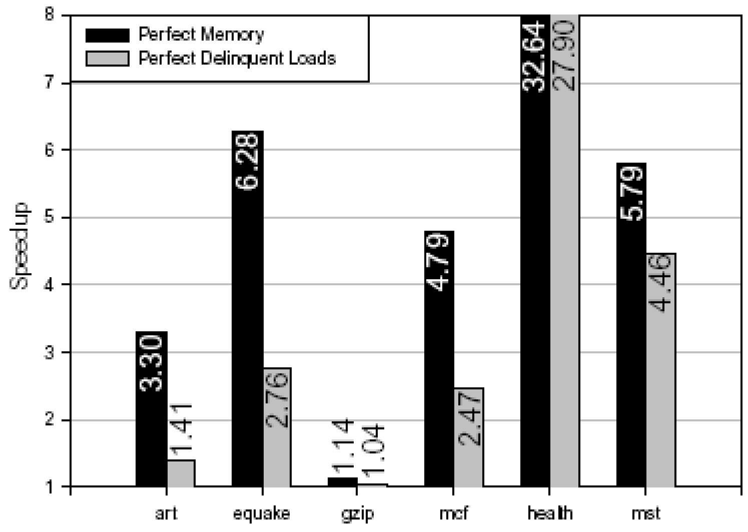


Figure 2. Speedup when 10 worst behaving static loads are assumed to always hit on cache (Courtesy of Jamison D. Collins [2])

4.2 *Delinquent loads*

For most programs, the cache misses are dominated by a few static loads. We call these poorly behaved loads delinquent loads. Figure 1 shows the cumulative L1 data cache misses contributed by the worst behaving static loads.

In most cases, eliminating performance losses from only the delinquent loads yields much of speedup achievable by the elimination of miss penalties. Figure 2 compares results when the worst 10 delinquent loads are assumed to always hit in the L1 cache, versus a perfect memory subsystem where all loads hit in the L1.

Identification of delinquent loads is determined by memory access profiling, performed by a compiler or a memory access simulator, that uses the total number of L1 caches misses as the criterion to select delinquent loads.

4.3 *P-Slices*

Precomputation slices (p-slices) are extracted from the program being accelerated. Once a load is identified as delinquent, a p-slice is constructed to prefetch the load. By eliminating instructions that delinquent loads do not depend on, the resulting p-slices are typical of very small sizes, typical 5 to 15 instructions per p-slice.

As a trigger instruction reaches some point in the pipeline, the corresponding p-slice is spawned into an available thread context. P-slices are spawned under one of two conditions: when encountering a basic trigger, which occurs when a designated instruction in the main thread reaches a particular commit stage, or a chaining trigger, when one speculative thread explicitly spawns another.

A speculative thread is spawned by allocating a hardware thread context, copying necessary live-in values into its register file, and providing the thread context with the address of the first instruction of the thread.

5 Software based SP

A machine that employs ideal, one-cycle flash copy between registers files of two thread contexts, permits the non-speculative thread to spawn speculative threads instantly without incurring any overhead cycles. An ideal machine may be difficult to implement on processors like the Itanium family processors, due to the cost of implementing a flash copy mechanism for such large register files.

In the case of Itanium, we can explore a less aggressive but more practical software-based SP (SSP) approach, which circumvents such hardware costs by directly taking advantage of the processor architectural features.

SSP requires two basic mechanisms to support thread spawning: a mechanism to bind a spawned thread to a free hardware context, and a mechanism to transfer necessary live-in values to the child thread. Its possible to implement both of these using existing features of the Itanium processor family: on-chip memory buffers, which are used as spill area for the backing store of the Register Stack Engine (RSE); and the lightweight exception recovery mechanism, which is used to recover from incorrect control and data speculations.

5.1 *Lightweight Live-in Transfer*

Processors in the Itanium family contains special on-chip memory buffers for use as backing store for the RSE to host temporary spilled registers. We call this buffer space the Live-in Buffer (LIB). We use this buffer space for passing live-in values from a parent thread to a child thread through normal loads and stores. The parent thread stores a sequence of values into LIB before spawning the child thread, and the child thread, right after binding to a hardware context, loads the live-in values from the LIB into its context.

5.2 *Lightweight Exception Recovery*

We can spawn a speculative thread and bind it to a free hardware context via the Lightweight Exception Recovery (LER) mechanism in the Itanium architecture.

LER uses speculation check instructions to examine the results of user level control or data calculations, to determine success or failure. Should failure occur, an exception surfaces and a branch is taken to a user defined recovery handler code within a thread, without OS intervention.

For example, a chk.a (advanced load check) instruction detects that if some store conflicts with an earlier advanced load occurs, it will trigger branching into a recovery code, and execute a sequence of instructions to repair the exception. Another example is the chk.c (available context check), that raises an exception if a free hardware context is available for spawning a speculative thread. Otherwise, behaves like a nop. The chk.c instruction is placed in the code wherever a basic trigger is needed.

5.3 *Performance of SSP*

Spawning a thread on SSP approach is no longer instantaneously as the hardware approach, and will slow down the non-speculative thread by the time necessary to invoke

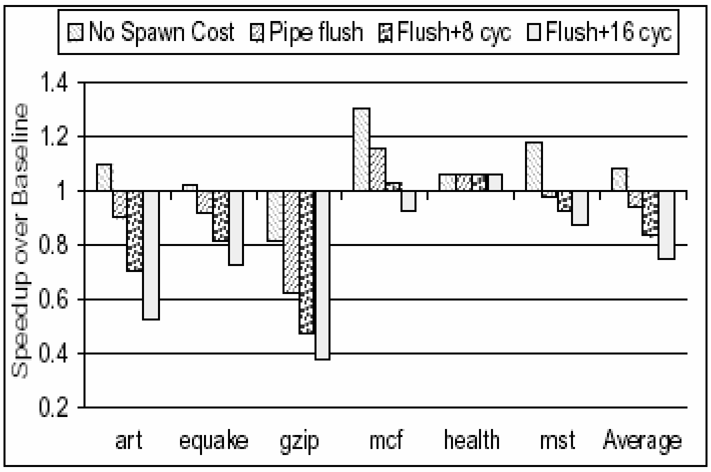
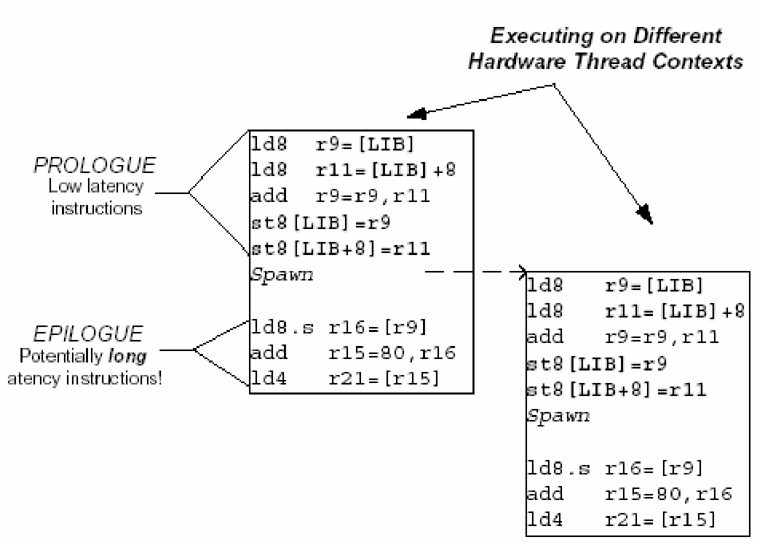


Figure 3. Speedup achieved by the software-based SP approach. Each bar corresponds to a different cost associated with spawning threads in the commit stage (Courtesy of Hong Wang [3])

and execute the exception handler, and the p-slices must be modified to first load their values from the LIB.

Figure 3 shows the speedup achieved by the SSP approach. In the second configuration from the left, the pipeline flush is the only penalty, while in the third and fourth configurations, an additional penalty of 8 and 16 cycles respectively is assumed for the cost of executing the recovery handler code.

A third option is the use of chaining triggers, a more aggressive form of SP that minimizes the overhead imposed on the non-speculative thread.


Executing on Different Hardware Thread Contexts
PROLOGUE
Low latency instructions
EPILOGUE
Potentially *long* latency instructions!

```
ld8   r9=[LIB]
ld8   r11=[LIB]+8
add   r9=r9,r11
st8[LIB]=r9
st8[LIB+8]=r11
Spawn

ld8.s r16=[r9]
add   r15=80,r16
ld4   r21=[r15]
```

```
ld8    r9=[LIB]
ld8    r11=[LIB]+8
add    r9=r9,r11
st8[LIB]=r9
st8[LIB+8]=r11
Spawn

ld8.s  r16=[r9]
add    r15=80,r16
ld4    r21=[r15]
```

Figure 4. Runtime behaviour of p-slice after being enhanced to incorporate chaining triggers

6 Chaining Triggers

Chaining p-slices are able to spawn future instances of themselves, decoupling thread spawning from non-speculative thread execution. This allows delinquent loads to be targeted far ahead of the non-speculative thread, but without significantly increasing executed speculative instructions.

P-slices containing chaining triggers typically have three parts: a prologue, a spawn instruction for spawning another copy of the p-slice and an epilogue. Figure 4 shows the runtime behaviour of a p-slice that incorporates chaining triggers.

The prologue consists of instructions that compute values associated with a loop carried dependency, i.e., those values produced in one loop iteration and used in the next loop iteration, such as updates to a loop induction variable. The epilogue consists of instructions that produce the address for the target delinquent load.

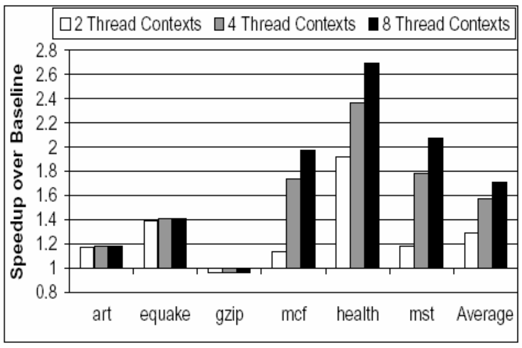


Figure 5. Speedup from Speculative Precomputation using both Basic and Chaining Triggers (Courtesy of Jamison D. Collins)

Figure 5 shows the runtime performance of a chaining p-slice. Particularly, in situations where there are loops, chaining triggers can advance to future loop iterations much faster than the non-speculative thread. This feature makes it possible to achieve dramatically higher performance than with basic triggers alone.

Spawning a trigger via a chaining trigger requires significantly less overhead than a basic trigger, because a chaining trigger requires no action from the main thread.

7 Conclusion

This paper presents Speculative Precomputation (SP), a technique that allows a multithreaded processor to use spare hardware contexts to spawn speculative threads to prefetch data well in advance of the main thread. When spawning threads fall on the main non-speculative thread (via basic triggers), the potential speedup is as high as 30% assuming fast register copies between thread contexts [2]. However, under more realistic assumptions, the potential speedup is significantly reduced. On the other hand, when the speculative threads can also spawn other speculative threads (via chaining triggers), dramatic speedups are possible on applications that have historically been resistant to prefetching techniques. These speedups are as high as 169% and average 76% over all benchmarks [2]. This is achieved via a novel software based mechanism that can utilize existing Itanium processor features with very little additional hardware support.

References

[1] Abraham, S.G, Rau, B.R.: Predicting load latencies using cache profiling, Hewlett Packard Lab, Technical Report HPL-94-110 (Dec. 1994).

[2] Collins, Jamison D., Tullsen, Dean M., et al: Speculative Precomputation- Long-range Prefetching of Delinquent Loads, 28th International Symposium on Computer Architecture (July 2001).

[3] Wang, Hong, Wang, Perry H., et al: Speculative Precomputation- Exploring the Use of Multithreading for Latency, Intel Technology Journal, Issue nº1- Vol. 6 (February 2002)

[4] Collins, Jamison D., et al: Dynamic Speculative Precomputation, 34th International Symposium on Microarchitecture (December 2001)

[5] Luk , C.: Tolerating memory latency through software controlled pre-execution in simultaneous multithreading processors, 28th Annual International Symposium on Computer Architecture (July 2001) 40-51