Putting Checkpoints to Work in Thread Level Speculative Execution

Salman Khan

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2010

Abstract

With the advent of Chip Multi Processors (CMPs), improving performance relies on the programmers/compilers to expose thread level parallelism to the underlying hardware. Unfortunately, this is a difficult and error-prone process for the programmers, while state of the art compiler techniques are unable to provide significant benefits for many classes of applications. An interesting alternative is offered by systems that support Thread Level Speculation (TLS), which relieve the programmer and compiler from checking for thread dependencies and instead use the hardware to enforce them.

Unfortunately, data misspeculation results in a high cost since all the intermediate results have to be discarded and threads have to roll back to the beginning of the speculative task. For this reason intermediate checkpointing of the state of the TLS threads has been proposed. When the violation does occur, we now have to roll back to a checkpoint before the violating instruction and not to the start of the task. However, previous work omits study of the microarchitectural details and implementation issues that are essential for effective checkpointing. Further, checkpoints have only been proposed and evaluated for a narrow class of benchmarks.

This thesis studies checkpoints on a state of the art TLS system running a variety of benchmarks. The mechanisms required for checkpointing and the costs associated are described. Hardware modifications required for making checkpointed execution efficient in time and power are proposed and evaluated. Further, the need for accurately identifying suitable points for placing checkpoints is established. Various techniques for identifying these points are analysed in terms of both effectiveness and viability. This includes an extensive evaluation of data dependence prediction techniques. The results show that checkpointing thread level speculative execution results in consistent power savings, and for many benchmarks leads to speedups as well.

Acknowledgements

I'd like to thank my adviser, Marcelo Cintra, for providing the guidance I needed, but also for letting me find my own path.

I would also like to thank all my colleagues and friends at *ICSA*. I have learned a great deal from them and they have made my time at Edinburgh enjoyable as well as productive. In particular Polychronis Xekalakis and Nikolas Ioannou, who showed exemplary patience while sharing an office with me.

I am grateful to Alasdair for his support and patience throughout my PhD and especially during the preparation of this document. Finally, I'd like to thank my parents who have always encouraged me and have enabled me to get this far. Without them, none of this would be possible.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Salman Khan*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Chip Multiprocessors and Parallelisation

Since the early days of microprocessors, designs have evolved from simple, microcoded processors to complex, wide superscalars with multiple levels of cache. This has led to steadily improving performance, but at the cost of extremely complex designs. In the last decade, it has become clear that adding complexity to single processors to achieve greater performance is providing diminishing returns. Meanwhile, the number of transistors available per chip has continued to grow. This has led to Chip Multiprocessors (CMPs) becoming the mainstream design choice for general purpose computing.

In the absence of coarse grained parallelism, the performance potential of Chip Multiprocessors remains unrealised. To improve performance of sequential programs on CMPs, attempts are made to extract thread level parallelism from the sequential program flow. This may be done manually by the programmer or through various compiler techniques. When a sequential program is parallelised conventionally, the programmer or compiler needs to ensure that threads are free of data dependences. If data dependences do exist, threads must be carefully synchronised to ensure that no violations occur. Specially for languages that support pointers, compile time disambiguation is often impossible. This means that compilers are often unable to extract much parallelism from sequential code while still guaranteeing correctness. In the absence of speculation, inter-thread dependences constrain performance and automated

thread partitioning seldom results in high performance (17).

Improving the performance of hard to parallelise programs through exploiting the opportunities presented by CMPs has been an area of intense research in recent years. This includes work in three broad categories. First, improved languages and programming models to allow programmers to more easily expose parallelism. Second, improved static analysis so that more parallelism can be extracted at compile time. And third, the area this thesis is concerned with, techniques to speculate on parallelism so that precise static dependence analysis is no longer required. With Thread Level Speculation (TLS), the compiler only needs to divide the code into threads. It does not need to guarantee that the multithreaded code is free of data dependences. If a dependence violation occurs at run-time, the speculation hardware detects the violation and rolls back the violating thread to a safe state. This mechanism allows the compiler to parallelise very aggressively and extract parallel threads from code that would otherwise not be parallelisable.

While Thread Level Speculation has been shown to provide significant performance improvements for hard to parallelise applications, questions remain about the efficiency of speculative execution. Re-execution of misspeculated portions of the program is wasteful in both time and energy. In case of a data dependence violation, TLS mechanisms (discussed in Section 2.1) lead to re-execution of *all* the instructions of the task in question, regardless of whether those instructions are dependent. The focus of this thesis is the reduction of wasteful re-execution. First, it proposes mechanisms for efficient checkpointing. Second, it uses dependence prediction as an effective way of placing checkpointing.

1.2 Contributions

1.2.1 Efficient Checkpointing

This thesis presents new ways of addressing the inefficiency of Thread Level Speculative execution by proposing an intermediate checkpointing scheme based on dependence prediction. The idea of checkpointing speculative tasks is not new, however,

previous work in the area of checkpointing speculative tasks has left many open questions. This is specially true about checkpoint placement policies. This thesis addresses implementation and policy details that are missing from previous work.

A detailed study is performed of the mechanisms involved in checkpointing and the policy issues that it exposes. Specifically, checkpointing is applied to a state-of-the-art TLS system that supports out of order spawning of speculative tasks. Observing task behavior leads to the conclusion that the base TLS protocol is not well suited to checkpointed execution. This motivates extensions in the TLS protocol to allow effective checkpointing. Further, a dependence predictor based policy is used to effectively place checkpoints.

1.2.2 Dependence Prediction

Dependence prediction is an important aspect of placing checkpoints effectively. Dependence predictors of varying levels of sophistication also have uses beyond checkpointing. They are important in synchronising speculative tasks to avoid data dependence violations (7; 42) and in resource management (53). This thesis proposes program counter and hybrid dependence prediction techniques in addition to previous address based ones, and performs a detailed evaluation of all these methods. It also discusses the complexity of constructing these various predictors.

1.3 Structure

This thesis is organised as follows.

Chapter 2 provides background on Thread Level Speculation and techniques for reducing unnecessary re-execution. This includes work in value prediction and in particular intermediate checkpointing. This chapter ends with details on the TLS hardware support assumed for the remainder of the thesis.

Chapter 3 proposes mechanisms for allowing efficient intermediate checkpointing. First, basic support for inserting intermediate checkpoints is described. Later, ways of making checkpointing more efficient are presented, including a modified restart mechanism and changes to the versioned memory system.

Chapter 4 looks at the policy options for inserting checkpoints. These include previously proposed stride checkpointing, and using dependence prediction to place checkpoints. It goes on to describe and compare various dependence predictors. The effects of overheads on checkpoint placement policy are also discussed.

Chapter 5 describes the simulator setup and the benchmarks used for evaluation. The metrics used for evaluation are discussed and justified. This includes metrics for dependence prediction and the performance of checkpointing schemes.

Chapter 6 performs a quantitative evaluation of the techniques described in this thesis. First, there is an extensive evaluation of dependence predictors and a comparison of different schemes. Then, the savings due to checkpointing are evaluated for various placement policies. The effects of the architectural extensions for efficient checkpointing proposed in this thesis are demonstrated. Finally, a brief demonstration is made of applying dependence prediction to synchronisation.

Chapter 7 looks at related work. This includes other uses of checkpointing, such as early recycling of resources and fault tolerance as well as other schemes for checkpointing speculative execution.

Finally, **Chapter 8** concludes this thesis by summarising the contributions and discussing avenues of future extension.

Chapter 2

Background

This chapter surveys the background for this work. The first part, in Section 2.1, provides background on work on Thread Level Speculation (TLS), looking at the execution model, the architectural support needed, compiler and task selection issues and the various systems proposed. This section also introduces the terminology associated with TLS, which is employed throughout the rest of the thesis. Then, in Section 2.2 various techniques to address the overhead of wasted re-execution in speculative execution are discussed. Synchronisation aims to avoid speculative overlap when dependences exist. Checkpointing reduces the re-execution required by changing the granularity of task units that need to be restarted on a violation. Value prediction techniques aim to avoid restarts by predicting values for speculative uses which would otherwise have caused a violation. Finally, in Section 2.3 the TLS mechanism assumed in the rest of the thesis is detailed.

2.1 Thread Level Speculation

In the absence of coarse grained parallelism, Chip Multiprocessors (CMPs) generally do not match the performance of superscalar processors of comparable die area. To improve performance of sequential programs on CMPs, attempts are made to extract thread level parallelism from the sequential program flow. This may be done manually by the programmer or through various compiler techniques. When a sequential pro-

gram is parallelised conventionally, the programmer or compiler needs to ensure that threads are free of data dependences. If a data dependence does exist, threads must be carefully synchronised to ensure that no dependence violations occur. In many cases, this cannot be effectively done by compilers. Specially for languages that support pointers, compile time disambiguation is often impossible. This means that compilers are often unable to extract much parallelism from sequential code while providing correctness guarantees. In the absence of speculation, inter-thread dependences constrain performance and automated thread partitioning seldom results in high performance (17).

Under the TLS execution model (also referred to in the literature as Speculative Multithreading), a sequential application is split into threads which are then speculatively executed in parallel with the hope that their concurrent execution will not violate sequential semantics (14; 16; 21; 38; 40). The control flow of the sequential code imposes a total order on the threads. A thread is said to be speculative if it is executing such that it is overlapped with a part of the program that is earlier in sequential order. Threads earlier in the sequential flow are termed less speculative (or *predecessors*) with respect to later threads (*successors*). The least speculative thread being executed is called the *head* or the *safe thread*. During speculative execution of threads, reads are monitored to detect data dependence violations. Writes may be forwarded from less speculative threads to more speculative ones in order to minimise violations. When executing tasks speculatively in parallel, correct behaviour is maintained by keeping the tasks ordered and making sure that no data dependences are violated. If a data dependence violation is detected, the consumer of the misspeculated data must be *squashed* along with its successors. A squash can result in a *restart*, reverting the state back to a safe position from which the thread can be re-executed. Alternatively, a thread can be *killed*, where the thread is simply destroyed and no re-execution is attempted. In most schemes a squash rolls the execution back to the start of the thread, but some proposals in the literature use periodic *checkpointing* of threads (8) such that upon a squash it is only necessary to roll the execution back to the closest safe checkpointed state. When the execution of a non-speculative thread completes it *commits* and the values it generated can be moved to safe storage. To maintain sequential semantics, threads

must commit in order. When a thread commits, its immediate successor acquires non-speculative status and is allowed to commit. When a speculative thread completes it must wait for all predecessors to commit before it can commit. After committing, the processor is free to start executing a new speculative thread.

2.1.1 Speculation Mechanism

To provide the desired memory behaviour, the data speculation hardware must provide at a minimum:

1. A method for detecting true memory dependences, in order to determine when a dependence has been violated.

2. A method for backing up and re-executing speculative loads and any instructions that may be dependent upon them when the load causes a violation.

3. A method for buffering any data written during a speculative region of a program so that it may be discarded when a violation occurs or permanently committed at the right time.

Typical ways of achieving these are described below.

2.1.1.1 Tracking Dependences and Detecting Violations

Typically, all data written and read by a task is tracked in order to ensure that any violations are detected. This may be done at different granularities. Some systems do so for each word, others for cache lines.

A write marks a location as dirty. If the size of the write is equal to the granularity of tracking, the location is marked as *protected*. This is done so that dependence violations are not flagged for values that are produced in the same task that consumes them. Any read is marked as an *exposed read* unless it is from a protected location. A data dependence violation occurs when a task writes to a location that has been read by a more speculative task with an exposed read.

Locations can be checked for dependence violations immediately upon a write or dependences can be checked in bulk at the end of a task. Hardware speculation

schemes usually perform the checks on each store, while software schemes usually do the checks when a task finishes (6).

2.1.1.2 Buffering State

Speculative tasks generate speculative writes which cannot be merged with the state of the system unless the task commits. These writes are stored separately, typically either in the cache of the processor running the task or in a dedicated speculative store buffer. If the task successfully commits, the state is merged with system state. If it is squashed before it reaches completion, buffered state is discarded. A task only commits if it completes execution *and* becomes non-speculative. This ensures that tasks commit in order, thus preserving sequential semantics. Garzaran et al. (13) provide a taxonomy of buffering approaches along with their respective advantages.

2.1.1.3 Data Versioning

Each task has one version of each datum. If a number of speculative tasks are running on a system, each has a different version of shared data. On commit, versions are merged into system state in task order.

Some proposals allow one version per processor (27), while others support multiversioned caches and hence allow a speculative task to execute on a processor even if commit is still pending for a previously executed task (34). Colohan et al. (8) do not use versioned memory at all in the first level of cache, instead relying on a multiversioned $L2$ cache. To allow efficient execution in the presence of shared data, speculative systems also forward shared data from earlier threads to later threads.

2.1.1.4 Register Passing

TLS systems vary in the degree of direct communication between processors. Multiscalar (38) allows direct communication between processors, allowing passing of live registers to freshly spawned tasks. Other proposals assume CMPs with no direct communication between processors and in these, all communication has to take place through shared memory. If all live registers are transferred from the parent task to

the spawned task, there are no complications regarding initial processor state for the speculative task. Otherwise, the issue needs to be addressed in the compiler or runtime by ensuring either that the newly spawned task does not rely on any registers or that register values are transferred.

2.1.1.5 Out-of-Order Spawn

In a TLS system, task ordering has to be maintained at all times. Some TLS proposals can only do this for in-order spawns. This means that tasks can only be created in the same order as sequential execution. This constraint can be enforced by only allowing the most speculative task to spawn another task. This means that each task can spawn at most one task. In-order spawn allows tasks to be spawned for only one loop level. Other systems support out-of-order spawning as well (32; 34). In this case, *any* task can spawn another speculative task. This allows nested tasks to safely be spawned.

Support for out-of-order spawn allows more parallelism to be extracted from nested loops, nested function calls, loops within function calls, etc.

2.1.1.6 Control Speculation

Most of the TLS architectures discussed support only data speculation. This means that speculation can only occur between points that are *execution equivalent* [1]. However, the Superthreaded Architecture (45) supports control speculation, but not data speculation. The Superthreaded compiler (44) can convert data dependences into control dependences, so that they can be enforced even without hardware data speculation support. Mitosis (20; 30) supports both control and data speculation. It does so by speculating between points that are not execution equivalent, but then adding *cancel* instructions on incorrect paths. This ensures that tasks that are created along misspeculated paths are killed.

[1] Two locations are said to be execution equivalent when one of them executes if and only if the other does, and they both execute the same number of times.

2.1.1.7 Spawn and Commit Mechanism

The hardware must provide a mechanism for starting tasks and for signalling that tasks have reached completion. Typically this is done through special *spawn* and *commit* instructions or through software control handlers supported by some specialised registers in hardware, as in the case of the Hydra CMP (27).

2.1.2 Compilation

Thread Level Speculative execution typically requires some compiler support. The compiler can be tasked with one or more of task selection, code generation and TLS specific performance optimisations.

In most proposed systems, task selection is done statically at compile time. One option is to use high level program structure to select tasks. This means constructs such as loops and function calls are candidates for tasks. Proposals that use this approach include the POSH (18) and Spice (31) compilers. Other systems take a more general approach. Mitosis (30) identifies *spawning pairs*, which are pairs of instructions that meet certain conditions of control and data independence. The Min-Cut approach to decomposition by Johnson et al. (15) applies graph theoretic algorithms to the control flow graph, such that all basic blocks and combinations of basic blocks are candidates for tasks. Other proposals that are not restricted to loops and functions for task selection include Multiscalar (48) and the compiler framework by Bhowmik and Franklin (1).

Not every candidate task performs well when speculated on. There are various ways of pruning out inefficient tasks. One possibility is to perform a preliminary task selection and then profile the resulting decomposition (18). Another is to leverage information about dependences between tasks at compile time. When taking static decisions on the quality of task decomposition, dependence relationships between tasks are important. In traditional, non-speculative parallelisation, if pointers are present pointer analysis needs to be performed to guarantee that there are no inter-task data dependences. This can be done by comparing the read and write references between them. Traditional pointer analysis techniques classify points-to relationships into those

that definitely hold and those that may hold. TLS does not require guarantees of tasks being dependence free, but would benefit in performance from information about the likelihood of dependences across tasks. This information can be provided through a dependence profile of the sequential program, or through *Probabilistic Pointer Analysis* (4; 37). Dou and Cintra (9) take a different approach, constructing a model to predict task runtime and then choosing tasks with predicted speedups. Recently, the use of hardware based performance counters to create speculative tasks at runtime has been proposed (19), removing the need for making static compile time and/or profile based decisions.

Further, in systems that do not support register communication between cores, the compiler must ensure correctness by communicating all values between tasks through memory.

Some compilers also perform TLS specific optimisations to make speculative execution more efficient. For instance, Zhai et al. (55) look at identifying dependent scalar use and define pairs and then aggressively scheduling the *USE*s late in the consumer task and the *DEF*s early in the producer task. This increases overlap when synchronisation is being used. In that work, the synchronisation is statically performed at compile time, but the same optimisations can apply to dynamic, hardware guided synchronisation and checkpointing schemes. Steffan et al. (41) have noted that small loop bodies can be made more TLS friendly through loop unrolling. The interaction of loop unrolling with speculative execution is further studied by Wang et al. (50). Software value prediction code may also be inserted at compile time, as by Mitosis (20) and the Superthreaded architecture (44). This is discussed in Section 2.2.3.

2.2 Reducing Wasted Re-execution

When there are no dependences between tasks, TLS works well to achieve parallelism by overlapping execution of sections of code (Figure 2.1a). However, in many cases violations occur and remove much of the overlapped execution (Figure 2.1b). Dependence violations incur significant overhead. There have been a number of techniques proposed to reduce this overhead. These are discussed below.

2.2.1 Synchronisation

Synchronisation aims to avoid dependence violations by serialising parts of execution such that values are not consumed before they are produced. This can reduce wasteful re-execution (7; 25; 42; 54; 55).

In the work of Zhai et al. (54; 55), a compiler based approach is used to synchronise scalar communication between tasks. The compiler identifies communicated scalar values, then inserts *wait* and *signal* instructions, each of which is associated with the scalar through an architected register. The *wait* instruction stalls until the value is produced in the previous task and communicated through a *signal*. In this proposal, since scalar values are explicitly communicated, correct execution depends on maintaining correct synchronisation. This can be achieved trivially by placing all the *signal* instructions at the end of a task and all the *wait* instructions at the start. This has the effect of serialising execution. To achieve overlap, each *signal* is placed as early as possible and the associated *wait* as late as possible. Also, to avoid deadlock, *wait* and *signal* instructions for each synchronised scalar must appear on every possible path. Further, an aggressive instruction scheduling algorithm is used to maximise overlap. The work has also been extended to memory resident values (56). In this case, since the underlying TLS mechanism ensures correctness, the optimisations can be more aggressive.

There have been a number of hardware techniques proposed as well for synchronising speculative tasks. The Multiscalar architecture (12; 25) uses an *Address Resolution Buffer* to automatically synchronise dependent load-store pairs.

The Multiscalar approach relies on very close coupling between processors. In a more general shared memory environment, different techniques have to be employed. Cintra and Torrellas (7) achieve synchronisation by associating states with cache lines. This state information is kept in a *Violation Prediction Table*. If a location sees violations, the system first tries to value predict, and if this fails it falls back on synchronisation. When a task consumes data from a cache line that commonly causes a violation and cannot be value predicted, it is stalled. At what point the consumer continues execution depends on the state of the cache line. In the *Stall&Release* state, it waits for the first write to the line. On the other hand, in the *Stall&Wait* state, it waits for all

possible writers, i.e. until the consumer becomes non-speculative.

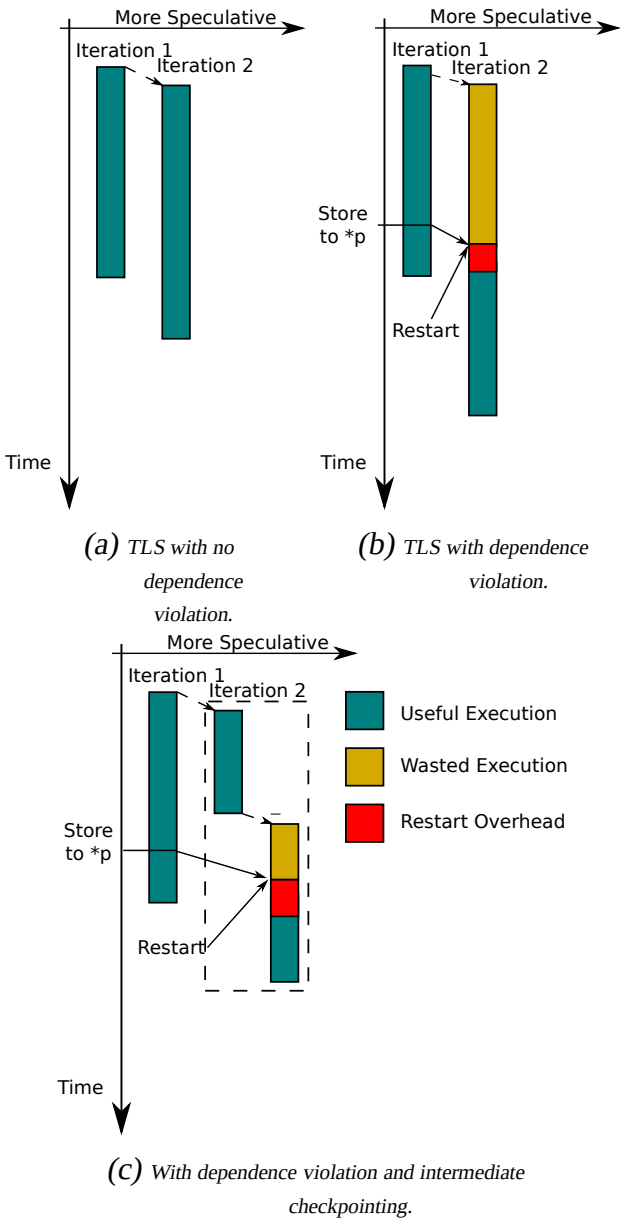
A slightly different approach is taken by Steffan et al. (42). This scheme also attempts to value predict in the first instance and falls back on synchronisation when the prediction confidence is low. However, instead of associating states with cache lines, it marks load *instructions* as being hard to predict. If a load instruction leads to violations, it is added to a *violating loads list*. This list is checked whenever a load instruction executes, and if the Program Counter of an executing instruction is found in the list, it is stalled. In this scheme, the load is always stalled until it becomes non-speculative.

2.2.2 Checkpointing

Intermediate checkpointing schemes aim to reduce misspeculation penalty by allowing partial rollback (Figure 2.1c). This is done by checkpointing the processor at some point or points during the execution of a task, and upon detecting a violation, only rolling back to the latest checkpoint which allows correct re-execution. Instead of avoiding violations, as synchronisation does, checkpointing aims to reduce the cost of violations.

The effect of checkpointing is quite similar to synchronisation when a violation does occur. In fact synchronisation is more efficient than checkpointing in cases where we can be sure that a violation will occur. On the other hand, if a violation occurs rarely, but at a high cost, then synchronisation may cause unnecessary serialisation, and checkpointing is a better alternative.

This can be understood better by inspecting the code snippet in Figure 2.3a. Assume that when the *cond* boolean is true, the resulting store in line 6 causes a squash to all subsequent threads, and that when *cond* is false, these threads commit without restart. In Figure 2.3b we can see what happens for the intermediate checkpointing case when speculation fails. The checkpoint is able to save some of the execution, however all the instructions executed after the checkpoint are wastefully executed. By synchronising instead, some power can be saved since the wasteful execution of these instructions is avoided (Figure 2.2c). However, when the store is not performed, and thus all threads commit, intermediate checkpointing is preferable. As is evident from



(a) TLS with no dependence violation.

(b) TLS with dependence violation.

(c) With dependence violation and intermediate checkpointing.

Figure 2.1: Example of Thread Level Speculation and checkpointing.

Figure 2.2d, we place a checkpoint before the load and proceed as normal. On the other hand, by synchronising on that load, the second thread will have to wait until it becomes safe, since the store which it attempts to synchronise with, is never performed by the less speculative thread (Figure 2.2e). This results in unnecessary serialisation.

From this example it is clear that from a performance point of view intermedi-

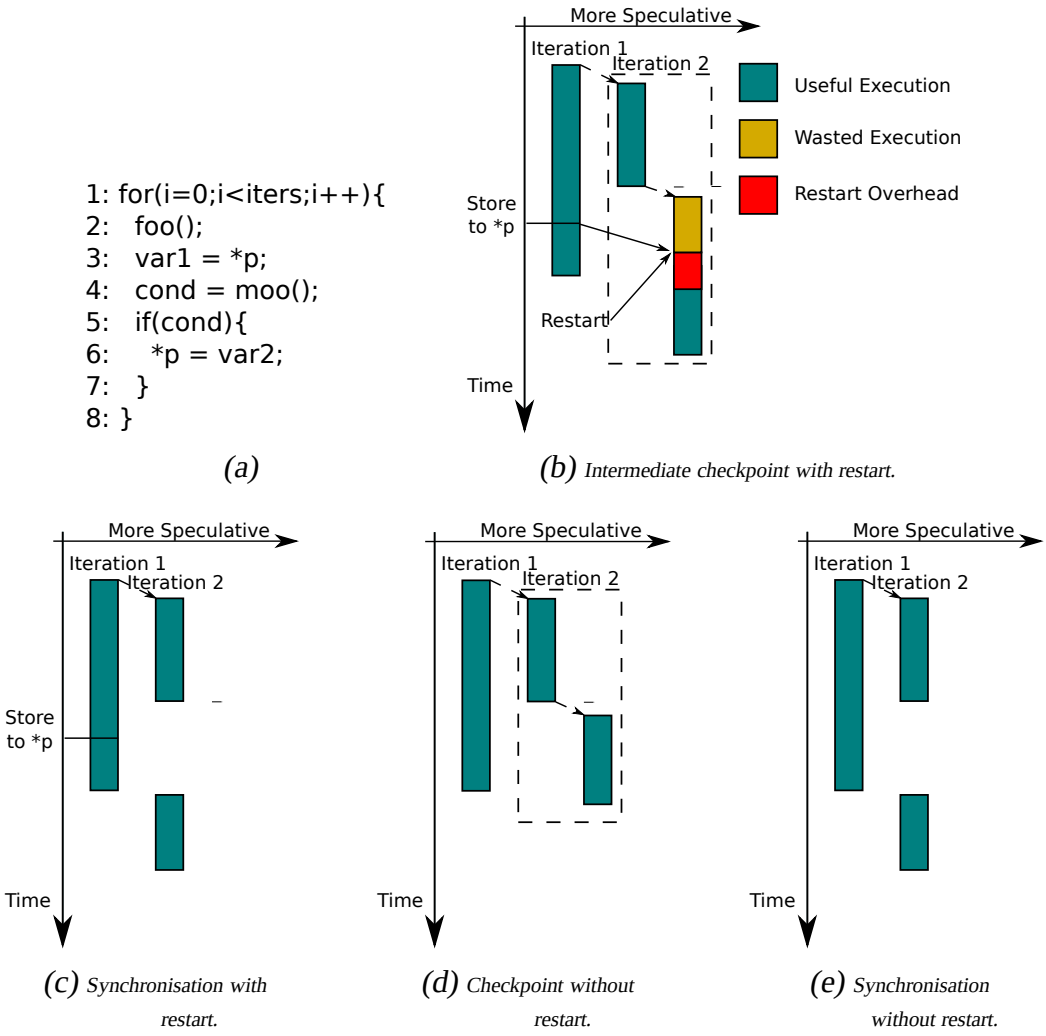


Figure 2.2: Comparing synchronisation and checkpointing: synchronising around predicted dependences can be problematic. We look at a code snippet: if *cond* is true, the store will restart subsequent tasks.

ate checkpointing is better (if we disregard the associated overhead of possible extra restarts, which is typically small). In fact, even threads that will have to be killed or restarted can indirectly provide performance benefits by prefetching for safer threads (34; 52). However, in terms of number of instructions executed, and thus energy consumed, synchronisation is typically better.

Checkpointing speculative tasks is studied by Colohan et al. (8). This is done in

the context of supporting efficient speculation for long running database tasks. The workload considered in that study consists of tasks that are often more than 50,000 dynamic instructions in size, and show large numbers of cross-thread dependences. The L2 cache is extended to maintain state for multiple thread contexts. A *sub-thread* is created by checkpointing register state and saving subsequent speculative state in the next thread context.

The decision to place a checkpoint can be taken in a variety of ways. Colohan et al. (8) place checkpoints periodically on fixed instruction strides. Waliullah and Stenstrom (49) place intermediate checkpoints in transactions in a Transactional Memory system [2], guiding checkpoint placement by identifying loads that may cause misspeculation. This is done by maintaining a *Critical Address Buffer*. Whenever a violation occurs, the address of the violating data is inserted into the buffer. The address of every speculative load is checked against this buffer and if it is found, a checkpoint is inserted. Checkpoint placement schemes are explored in detail in Chapter 4.

2.2.3 Value Prediction

There have been a number of proposals that include prediction of speculative values so as to avoid dependence violations. Value prediction has been suggested in both software and hardware. Successfully predicting values communicated from less speculative tasks to more speculative ones breaks dependences altogether and allows for more overlap. The Spice proposal (31) splits loop iterations into as many speculative chunks as there are cores available, and inserts code for predicting live-ins for each chunk. The POSH compiler (18) performs value prediction as well, but only for what it identifies as induction variables. Mitosis (20; 30) adopts a much more general method for value prediction, inserting pre-computation slices at the start of speculative sections. This is done by traversing the control flow graph backwards starting at the speculative section. Instructions that produce the live-ins to the speculative section are selected. These selected instructions are then summarised using profile information. For instance, rarely taken control paths are pruned out, as are instructions that the live-ins are infrequently

[2] Transactional Memory (TM) is a speculative parallelisation technique related to TLS. TM systems also suffer ineffiency due to wasteful re-execution.

dependant on. These summarised instructions are then duplicated at the start of the speculative section to form a *p-slice*. The live-ins produced by these *p-slices* are validated when the previous task has both ended and become non-speculative.

A number of proposals incorporate hardware value prediction for both register and memory resident values. Cintra and Torrellas (7) propose a framework for learning and predicting violations, and using value prediction in certain states. They only evaluate a simple last value predictor and find that it does not improve performance significantly. Steffan et al. (42) have a similar scheme that throttles value prediction and only employs it under certain circumstances. Values are predicted when the load in question is likely to squash *and* the prediction confidence is high. If the prediction confidence is low, the load is synchronised. This is discussed in Section 2.2.1. The scheme is evaluated with an aggressive hybrid context and stride predictor.

Marcuello et al. (22) evaluate various value predictors on their Clustered Speculative Multithreaded processor. They look at innermost loops in SPECint95 and propose a new predictor targeted specifically toward speculatively multithreaded systems, called a *new value predictor*. They conclude that it is beneficial to value predict register dependences, but that memory value prediction did not lead to much further improvement.

Prior to this, Oplinger et al. (28), as part of a study to identify sources of potential performance improvement using speculative multithreading, observed that return value prediction for procedures and stride value prediction for loops can improve performance.

While synchronisation is an *alternative* to value prediction, checkpointing and value prediction can be *combined*. If a checkpoint is placed when a value is predicted, the misprediction penalty is reduced. Unlike synchronisation, checkpointing does not result in a stall.

2.3 Hardware Support for Speculative Multithreading

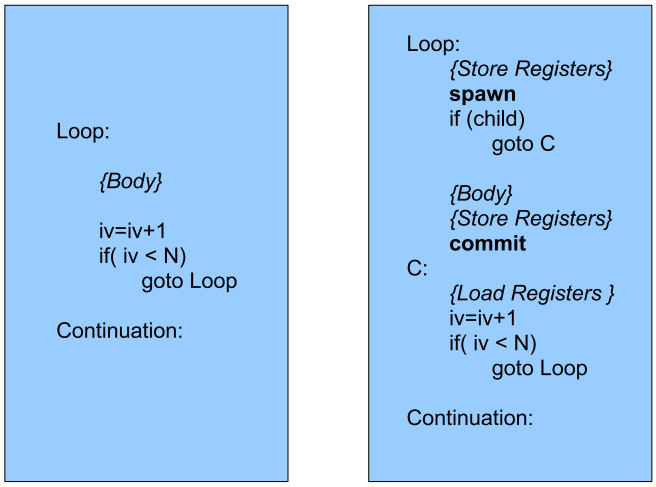
In this section, the details of the TLS execution model and the hardware to support it are described. These form a baseline model for the rest of the thesis.

The basic TLS model used is very similar to the one proposed by Renau et al. (32; 34). Data versioning and dependence tracking are handled through having a multiversioned $L1$ cache. If a task attempts to perform a speculative load and there is no space in the $L1$ cache to allocate a speculative line, the task cannot continue. In this case, the most speculative task on the processor is restarted in an attempt to free some speculative memory. The $L2$ cache is shared and only contains non-speculative data. Spawns and commits are performed through explicit instructions. When a *spawn* instruction is encountered, a new task is spawned off on a different processor, with the Program Counter, Stack Pointer and some task ordering information copied over. No other registers are copied, and hence the compiler is responsible for spilling registers around spawn instructions and ensuring that live-ins for spawned tasks are communicated through memory. Since both the parent and child task start with the same Program Counter, a mechanism needs to exist for each task to execute the correct code. This is done through the spawn instruction returning different values. That is, the standard return value register (r31) holds 0 in the parent and 1 in the child after the spawn. A conditional branch placed by the compiler after the spawn instruction can thus choose the correct code to execute. An example of such a spawn for loop level speculation is shown in Figure 2.3.

Since out-of-order spawn is supported, there are no restrictions on which tasks are allowed to spawn. This support is through splitting task ID ranges as proposed by Renau et al. (34).

During speculative execution, whenever a task performs a store, it is immediately written through the $L1$ cache to the bus, and becomes visible to all other processors. If, for any task more speculative than the task performing the store, the address matches an exposed load, the more speculative task is immediately restarted. Once a task reaches the end of execution (executes a *commit* instruction), it becomes ready to commit. A task cannot actually commit state until it becomes non-speculative. The commit process itself involves propagating all speculatively written data to safe system state [3], followed by passing the *commit token* to the next more speculative task. This in turn

[3] This is done by writing all speculative $L1$ lines associated with the committing task to the $L2$ cache, and marking those lines in the $L1$ cache to be non-speculative.



(a) Original loop. *(b) With spawn inserted.*

Figure 2.3: Example of a simple loop spawn insertion. When the spawn instruction executes, both tasks start at the next instruction. The parent has *child* set to false and the child has it set to true.

informs the next task that it is now non-speculative and is allowed to commit.

Restarts are handled by restoring the Program Counter and Stack Pointer values with which the task was spawned and starting execution. Since speculative tasks are not allowed to assume live-ins through registers, the entire register file does not need to be restored.

As an energy optimisation, the number of restarts is limited. If a task receives a violation and restarts three times, it then stalls until it becomes non-speculative before it can continue. This is to prevent tasks with many dependences from wasting too much energy.

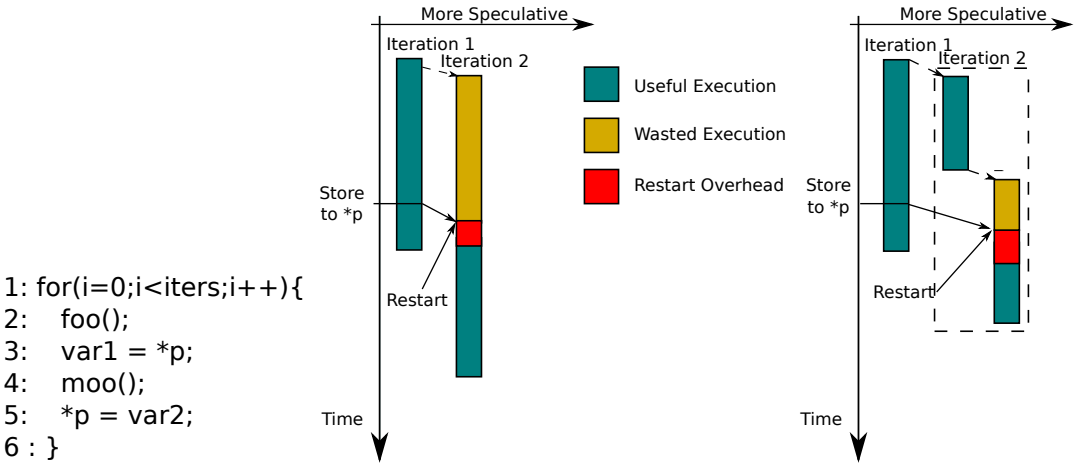
Chapter 3

Checkpointing Mechanism

Intermediate checkpointing of speculative tasks requires some architectural support. This chapter describes the mechanisms required to support efficient intermediate checkpointing. First, in Section 3.1, the basic checkpointing support is described, and the hardware requirements enumerated. In Section 3.2, cases are identified in which the basic scheme leads to inefficiencies, and extensions to overcome these problems are described.

3.1 Creation of Checkpoints

The insertion of checkpoints in the proposed scheme is quite straightforward. When a task is to be *checkpointed*, we simply spawn a new task which is an immediate successor to it. This is a hardware initiated spawn, unlike the compiler inserted spawn instruction mentioned in Section 2.1.1.7. We shall refer to this new task as the *checkpoint*. This process is shown in Figure 3.1, considering a simple case of two iterations of a loop, one of which is executed speculatively. In the base TLS case, the task receiving a violation restarts. In the checkpointing case, a checkpoint is inserted by spawning off a new task. We assume that the checkpoint is inserted just before the read from **p* in line 3. This *checkpoint* then behaves exactly as normal, and restarts upon receiving a violation. The difference is that the *checkpointed* task does not see a violation.



(a) Speculatively parallelised code *(b) Base TLS* *(c) Checkpointed*

Figure 3.1: Intermediate checkpointing showing two iterations.

Further, we take a snapshot of the register file so that the checkpoint can be restarted[1]. We constrain the newly created checkpoint to remain on the same processor as the checkpointed task. This simplifies the protocol and keeps overheads low because live registers remain available in checkpointed execution. If we allowed checkpoints to be started on a different processor, we would need complex support for communicating live registers across processors. Keeping the checkpoint on the same processor is also the obvious route to take with respect to data locality. Apart from pinning it to the same processor as the parent, the checkpoint is treated as any other speculative task. Any violations affecting the checkpoint only cause a restart to the checkpoint. If a violation affects the checkpointed task, it is restarted and since the checkpoint is a successor task, the checkpoint is killed. Any task can be checkpointed, even if it is a checkpoint. The versioned memory mechanism ensures that tasks have the correct state just as with normal tasks. It is worth observing that if the underlying TLS protocol only allows in-order tasks, only the most speculative task can be checkpointed through this mechanism. Since we use a base protocol that allows out-of-order spawn,

[1]In the base TLS protocol, no such snapshot is required when spawning a new task, since it is assumed that there are no live registers and all communication to the new task is through memory. So for non-checkpoint tasks, we only need to store the Program Counter and Stack Pointer in order to allow restarts, as discussed in Section 2.3.

we can checkpoint any task, whether it is the most speculative or not. However, as we demonstrate in Section 3.2 through examples, there are some extensions required to the TLS protocol to allow checkpointing to be effective in improving performance and reducing power consumption.

3.1.1 Hardware Requirements

The hardware changes required to allow checkpointing speculative tasks are minimal. The only difference between a normal spawn and a checkpoint is that a checkpoint requires a snapshot of register state. This can be done either by sets of shadow registers in the processor, or by storing these snapshots in memory. If the former route is chosen, snapshots can be taken very quickly, perhaps with no time penalty at all, but the number of checkpoints is limited. Such shadow register files are already supported in processors to allow recovery from branch mispredictions. If, on the other hand, registers need to be transferred to memory for each checkpoint, there is an associated latency. Storing the snapshot in memory need not cause a delay, since this process is not on the critical path and can be buffered and performed lazily. However, rewinding to a checkpoint would incur memory access latency in this case.

It is also possible to envision a hybrid process, which would keep register state for some checkpoints on the processor but allow older checkpoints or those less likely to be used to be moved to memory. If those checkpoints that are more likely to be needed can by successfully identified, this approach could give the best of both worlds.

For the purpose of evaluation in this thesis, checkpointing through shadow register files in the processor is modeled. The effects of supporting varying numbers of checkpoints are evaluated in Section 6.2.

Checkpointing a task and continuing execution of the checkpoint on the same processor presents the question of *when* the checkpoint begins execution. If, on a modern pipelined processor, we wait for the instructions from the checkpointed task to drain from the pipeline, there is a significant cost. There is no need to wait, however, and checkpoint instructions can follow immediately. In fact, once a decision to checkpoint is made, it is even possible to insert a checkpoint at an instruction already in the pipeline. It is important that loads and stores on either side of the checkpoint bound-

ary are marked with the correct task ID in the versioned cache. Also, the register file must be checkpointed at the state after the last instruction of the checkpointed task in sequential execution. This can be achieved by performing the checkpoint in two stages. Once the instruction where the checkpoint is to be placed is identified, mark all memory operations after that in sequential order with the child task ID. Second, when the first instruction in the checkpoint task is ready to commit, take a snapshot of the register file to associate with the checkpoint.

Apart from the mechanism for inserting checkpoints, there has to be hardware support for deciding *when* to insert a checkpoint. This can be done either by having explicit checkpoint instructions or by making the decision at runtime. Policies for checkpoint insertion and the hardware requirements for employing each policy are left for discussion in Chapter 4.

3.2 Efficient Checkpointing

Though the minimum support for inserting checkpoints is quite simple, as described above, observing checkpointed execution reveals that this may not be sufficient for checkpointing to be *efficient* in terms of execution time and power consumption. This is demonstrated through examples in the remainder of this chapter and hardware extensions for overcoming these shortcomings are proposed.

3.2.1 Selective Kills and Restarts

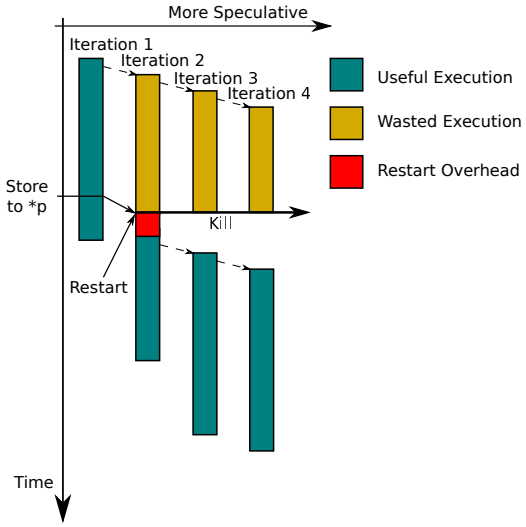
In TLS execution, a task may spawn a more speculative task and later get killed or restarted. In the base TLS mechanism, when a task is restarted, all the tasks that are more speculative are killed. However, since the task re-executes from the beginning, any tasks it spawned earlier (before restart) are now respawned[2]. Rewinding to a checkpoint may change this behaviour.

We look at this issue by revisiting the program in Figure 2.2. This time, in Figure 3.2, we look at speculating on multiple iterations of the loop, assuming a four

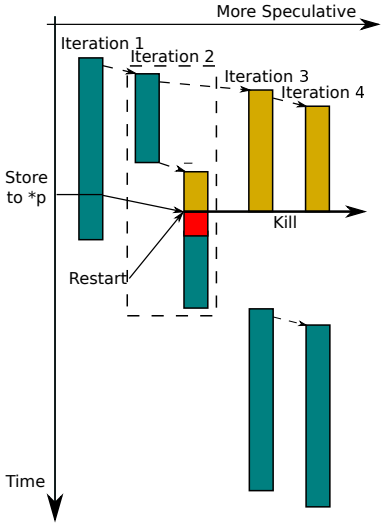
[2]This assumes that the spawns are on a control path that is taken each time. If this is not true, the tasks spawned before and after restart may be different.

```
1: for(i=0;i<iters;i++){
2:    foo();
3:    var1 = *p;
4:    moo();
5:    *p = var2;
6: }
```

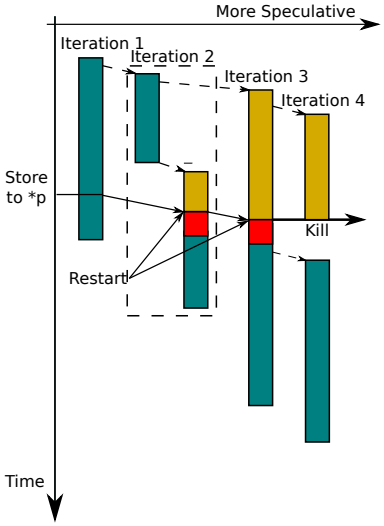
(a) *Loop that is speculatively executed.*



(b) *Base TLS*



(c) *Checkpointed*



(d) *Checkpointed with selective restart*

Figure 3.2: Intermediate checkpointing with four processors and a dependence violation. For simplicity only the first dependence violation is shown, and checkpoints are only shown for the first speculative task.

processor system. In Figure 3.2b, we look at the case without checkpointing. We see that a violating store to *p* in the safe task 1 (running iteration 1) causes task 2 (running iteration 2) to be restarted and tasks 3 and 4 to be killed. However, the restarted task 2 quickly causes new tasks to be spawned. Now, looking at the case in Figure 3.2c, where the speculative task 2 is checkpointed immediately before the offending load, we see that we save re-execution by only rewinding to the checkpoint. In this case, the processors that are left idle because tasks 3 and 4 are killed remain idle until task 2 finishes executing.

Checkpointing causes this behaviour often since when tasks are not rolled back completely, they do not necessarily respawn child tasks that have been killed. This is specially true of loop iteration speculation where spawns are often at the very start of tasks. This can cause or exacerbate load imbalance. This issue can be *rectified* with some changes to the TLS protocol, which are described below.

When a task is restarted, we have to be careful to maintain correctness in more speculative tasks. In the base protocol, this is achieved by simply killing all tasks more speculative than a task that is restarted. This is shown in Figure 3.3a. The algorithm there shows that whenever a task receives a restart, it restarts itself, and propagates a kill signal to the next more speculative task[3]. We observe that this is excessively conservative. The only tasks that have to be *killed* are those that were spawned incorrectly. That is, those tasks that were spawned by execution that turned out to have misspeculated. So, in the updated restart algorithm, we kill each task whose parent (spawning) task has been restarted or killed. Or, to state it differently, for each task that is restarted or killed, all its children are killed. For other tasks that are more speculative than any killed or restarted task, even though they were spawned correctly, it is still possible that they have consumed invalid values forwarded from less speculative tasks. Therefore, we restart all other tasks that are more speculative than any killed or restarted task. This modified restart mechanism is shown in Figure 3.3b. Here, the algorithm is expressed recursively. *inMerge* is used to check if a task's parent has been killed or restarted as part of the current chain of restarts and kills. Each task

[3] As described in Section 2.1.1.5, in out-of-order speculation the successor of a task is not necessarily its child, and the predecessor not necessarily its parent. This distinction should be kept in mind during the discussion of restart and kill mechanisms.

```
restart(task){
   restartProcessing(task);
   kill(task.next);
}

kill(task) {
   killProcessing(task);
   kill(task.next);
}
```

(a) Restart in base protocol.

```
restart(task){
   task.inMerge = true;
   if(task.next.parent.inMerge)
   kill(task.next);
else
   restart(task.next);
   task.inMerge = false;
}

kill(task) {
   task.inMerge=true;
   killProcessing(task);
   if(task.next.parent.inMerge)
      kill(task.next);
   else
      restart(task.next);
   task.inMerge=false;
}
```

(b) Updated restart.

```
restart(task){
   task.inMerge = true;
   if(task.startTime
         > earliestRestartTime) {
      earliestRestartTime =
         max(earliestRestartTime,
      task.startTime);
      restartProcessing(task);
   }
   if(task.next.parent.inMerge)
      kill(task.next);
   else
      restart(task.next);
   task.inMerge = false;
}

kill(task) {
   task.inMerge=true;
   earliestRestartTime =
   max(earliestRestartTime,
      task.startTime);
   if(task.next.parent.inMerge)
      kill(task.next);
   else
      restart(task.next);
   task.inMerge=false;
}
```

(c) Updated restart with timestamp comparison.

Figure 3.3: Restart algorithm showing how restart and kill events are recursively propagated.

that is restarted or killed sets *inMerge* to true before passing on a kill/restart token. For any task, *task.next* is the immediately more speculative task. *task.parent* is the parent or spawning task. By checking the value of *inMerge* for the parent of each task, we can ensure that all tasks that were spawned from misspeculated execuation are killed. It is worth noting that the same effect can be achieved by associating a list of task IDs with the kill or restart signal, with the ID of each task being appended to it when it is killed or restarted.

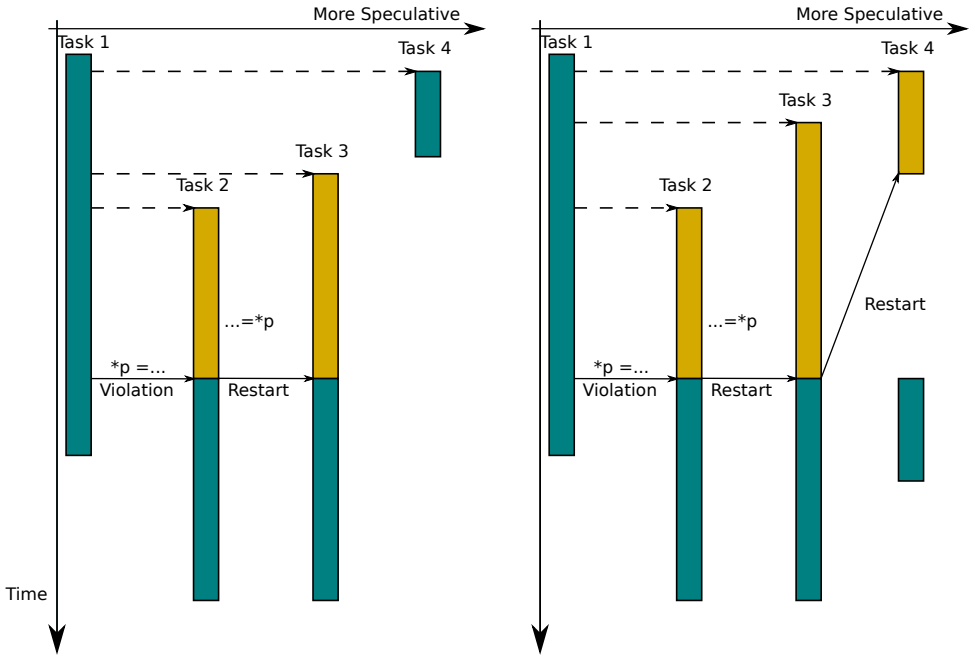
This issue is specific to out-of-order spawn, since if spawns are only in-order, for any given task, all tasks that are more speculative have been spawned by it or its successors. For in-order spawn, the algorithm in 3.3b reduces to killing all tasks more speculative than the restarted one.

If the new restart algorithm is used, we get the situation in Figure 3.2d. The checkpoint for task 2 gets restarted, but the checkpointed task does not receive a restart. This means that the task for iteration 3 now receives a restart instead of a kill since its parent is not in the restart/kill chain, and it immediately spawns off a new task 4. We see that checkpointing is no longer causing processors to remain idle.

Going further, not all more speculative tasks need to even be restarted. A given task requires a restart only if it has overlap with misspeculated execution[4]. This observation has previously been made by Colohan et al. (8). In that work, timestamps are tracked to compare the start time of a task receiving a violation with the end time of more speculative tasks. A more speculative task is only restarted if its end time is later than the start time of the task receiving the violation.

We now extend the restart/kill mechanism to keep track of task start and end times. To maintain a temporal ordering between tasks, each task is annotated with a *timestamp*, which is the value of the system clock when the task begins execution. Since the system studied here is a CMP with closely coupled processors, it is assumed that real time is available to each task, and hence a total ordering can be maintained. In situations where a total ordering cannot be established, the notion of *logical time*, as commonly used in distributed systems, can be employed to establish partial orderings

[4]Strictly speaking, a restart is only required if an incorrect value was forwarded from misspeculated execution. The scheme can be made more precise by tracking forwarded values but we avoid the complexity of that here.



(a) Task 4 does not require a restart because it has no overlap with misspeculated execution.

(b) Task 4 needs to be restarted because there is no guarantee that incorrect values are not consumed.

Figure 3.4: Example of selective restart using timestamps.

between tasks.

When a task receives a dependence violation and restarts, a kill/restart signal is propagated. Any task receiving the signal has to be killed if its parent was killed/restarted, and otherwise restarted only if it has overlap with a task that was killed or restarted. It is important to note that a timestamp comparison with only the task originally receiving the violation is *not* sufficient to guarantee correctness.

This restart requirement has to be enforced because in order to *maintain correctness*, we need to ensure that no invalid forwarded values can be kept. Since the correctness of forwarded values is not guaranteed until the source task of the value commits, the start time of any task needs to be compared with *all* predecessor tasks that get killed or restarted, not just the task receiving a violation[5]. When the restart/kill signal

[5]The work by Colohan et al. (8) appears to only perform a timestamp comparison with what they call a *primary violation*, which is the task receiving the dependence violation. This can cause incorrect execution, as shown in Figure 3.4b. To guarantee correctness, timestamp comparisons need to be performed recursively for more speculative tasks for each restarted task in the chain.

is propagated, each task is checked against the earliest start time of any task that has been killed or restarted in the chain. If there is any overlap, that is, the end time of the task in question is later than the start time of any killed/restarted task, the more speculative task is restarted. Otherwise nothing needs to be done. This is demonstrated in Figure 3.4. In Figure 3.4a Task 4 does not have any overlap with either of the restarted tasks 2 or 3, and so does not need to be restarted. On the other hand, in Figure 3.4b, a different situation is shown, where Task 4 is overlapped with Task 3. If Task 3 is restarted, there is no guarantee that the values consumed by Task 4 are correct, and Task 4 must be restarted. It should be noted that it is possible for Task 4 to consume incorrect speculative values even though it does not have any overlap with the task originally receiving the data dependence violation (Task 2). The algorithm for selectively restarting tasks using timestamps is shown in Figure 3.3c.

The effects of selective restart on checkpointed execution are evaluated in Section 6.3.2.

3.2.2 Memory Optimisation

The advantage of treating checkpoints like any other speculative task is that the TLS protocol needs very little change to support checkpoints. The primary disadvantage appears in increased pressure on versioned memory. Each checkpoint has to maintain a version of any speculative data it uses or produces. In the base TLS protocol, this means allocating a speculative line for every block that has been read by the checkpoint. When no space is available in the $L1$ cache and allocation fails, the most speculative task on the processor is restarted in attempt to free some speculative space. Our experiments show that this causes many restarts. Data locality makes it likely that a checkpoint will read locations read by its parent. This results in duplicate versions of data. Colohan et. al. (8) avoid this issue by making the $L2$ cache multiversioned, and by having no versioning in the $L1$ cache.

To deal with this problem, we propose some changes to the memory protocol. These are based on a certain relationship between a checkpointed task and its checkpoint. Specifically, it is guaranteed that a checkpointed task will have *no* overlapped execution with its checkpoint, and that the checkpoint will be the immediately more

speculative task. With these in mind, we can relax some of the constraints of the data versioning protocol. If the checkpoint accesses a location that is marked as an exposed read by its parent, it does not need to allocate a line for it. This is because if that read turns out to be a misspeculation, the parent will be restarted, killing the checkpoint. Because there is no overlap between the checkpointed task and its checkpoint, any misspeculation will be because of a store from a task less speculative than the checkpointed task. The checkpointed task cannot be a source of misspeculation in the checkpoint. Since the checkpoint is pinned to the same processor as the checkpointed task, there is no performance related reason to allocate a line for such accesses.

Note that stores still need to be buffered separately for the checkpoint and its parent in order to allow partial roll back, so checkpointing is not completely free of overhead in speculative state buffering.

The effects of these changes on checkpointed execution are evaluated in Section 6.3.1.

Chapter 4

Checkpoint Placement Policy

The important policy question when checkpointing tasks is where to insert checkpoints. Checkpoints should be placed so as to minimise wasteful re-execution. This means, ideally, placing a checkpoint just before any load that violates. Since in any realistic model of the system, a checkpoint will have some overhead, the insertion of checkpoints has to be made while taking into account resource constraints. In this chapter, the policy problem of inserting checkpoints is addressed in parts. First, in Section 4.1 the initial problem of identifying potential locations for checkpoints is addressed. Section 4.1.3 approaches this problem by looking at various ways of predicting dependences including address based, program counter based and hybrid schemes. In Section 4.2 the resource constraints of the system are analysed and discussed. Section 4.3 brings these issues together to propose a policy for inserting checkpoints. Finally, Section 4.4 looks at the viability of checkpointing policies and how they interact with the instruction pipeline.

4.1 Identifying Checkpoint Locations

4.1.1 Static Checkpoints

One possibility for placing checkpoints is to have specialised instructions to insert checkpoints. This would rely on static analysis to insert checkpoints at appropriate locations. Techniques similar to those used for synchronising dependences (55) can be

used. Other techniques, such as probabilistic pointer analysis (4; 37) can be useful for checkpoint insertion as well. This thesis concentrates on runtime techniques and does not evaluate static checkpoint insertion.

4.1.2 Stride Checkpoints

This is the simplest of the dynamic checkpointing policies evaluated in this thesis. Stride checkpointing involves inserting a checkpoint every N instructions. This method has been proposed by Colohan et al. (8). That work looks at large speculative tasks composed of database transactions, explores different stride values but does not consider any other method of placing checkpoints. The only additional hardware required for placing checkpoints by instruction stride is a counter to keep track of the number of instructions since the start of a task or since the previous checkpoint.

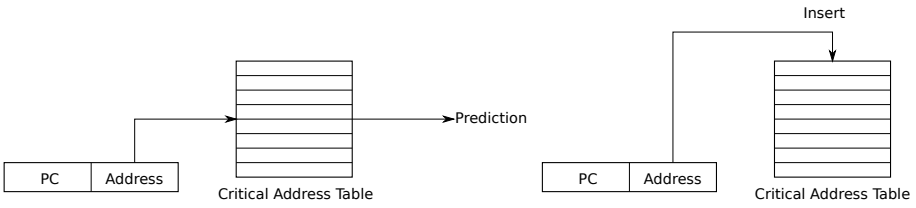
4.1.3 Checkpointing By Predicting Dependences

To make checkpointing more effective, checkpoints need to be placed intelligently before and as close as possible to violating loads. A perfect checkpointing scheme would place a checkpoint just before every violating load and nowhere else. At runtime, it is not possible to know with certainty which loads will lead to dependence violations, so prediction has to be employed. Hence we look at methods for predicting which loads are likely to violate, and use this information for placing checkpoints. Many dependence relationships *can* be known statically at compile time, and these can be synchronised. This does not preclude using dependence prediction for checkpointing to deal with dependences that are irregular or not analysable.

When studying dependence prediction for checkpointing, it is important to keep in mind the relative costs of mispredictions. As long as checkpoints can be placed with fairly low overhead, predicting violating loads is more important than total accuracy. In other words, the cost of a False Negative is higher than that of a False Positive. This, coupled with the observation that many dependences are infrequent and irregular, means that directly using counter based bimodal tables such as those typically used in branch predictors is not an appropriate choice.

4.1.3.1 Address Based Prediction

Waliullah and Stenstrom (49) looked at Transactional Memory systems and proposed inserting checkpoints before loads from addresses that have been seen to violate previously. A similar approach was used by Cintra and Torrellas (7) for synchronising dependences.



(a) To make a prediction the address is checked against the Critical Address Table.

(b) On a violation, the address is inserted into the Critical Address Table.

Figure 4.1: Address Predictor in action.

The address based predictor used in this thesis is similar to the one used by Waliullah and Stenstrom (49). Every time a violation occurs, the address of the violating store is added to a *Critical Address Buffer*, as shown in Figure 4.1b. On every load, the buffer is checked to see if the load address is critical (4.1a). If the address is found in the buffer, it is potentially violating and a checkpoint can be inserted. The size of this buffer can be kept quite small without losing much prediction accuracy. The effect of the size of the buffer is evaluated in Section 6.1.1. A prediction table is added to each processor. Compared to having a single system-wide buffer, this fragments the training history and leads to slower training. However, per-processor tables are used in order to keep prediction latency low. Latency also precludes adding dependence prediction information to existing per-line speculative state, as is done by Cintra and Torrellas (7). Since speculative state is kept in the $L1$ cache in the system modeled, such an approach would add the equivalent of an $L1$ access on every prediction. The latency issue is further addressed in Section 4.4.

A replacement policy for the buffer needs to be chosen. In Section 6.1.1, we evaluate random, Least Recently Used (LRU) and First-In-First-Out (FIFO) replacement policies. The replacement policy can make a substantial difference to prediction accu-

racy for small buffer sizes.

4.1.3.2 PC Based Prediction

Address based checkpointing works well in most cases, but there are some commonly seen patterns in programs where it fails to identify dependant loads. These include sliding array operations and pointer chasing. Examples are shown in Figure 4.2. In Figure 4.2a, there is a dependence through a member of a different object pointed to through *p* for each iteration. Figure 4.2b shows a loop carried dependence through a different element of array *A[]* for each iteration. Both of these constitute loop carried dependences that are not predictable by address, since each instance of the dependence is through a different address. It is clear, however, that in this example, there is a pattern to the dependences which should be predictable. For cases such as this, a prediction can be made using the violation history of *instructions* rather than memory addresses. A similar approach was used by Moshovos et al. (25) and Steffan et al. (42) for synchronisation.

```
while(p){
  foo();
  z = p->count;
  moo();
  p=p->next;
  p->count++;
}
```

(a)

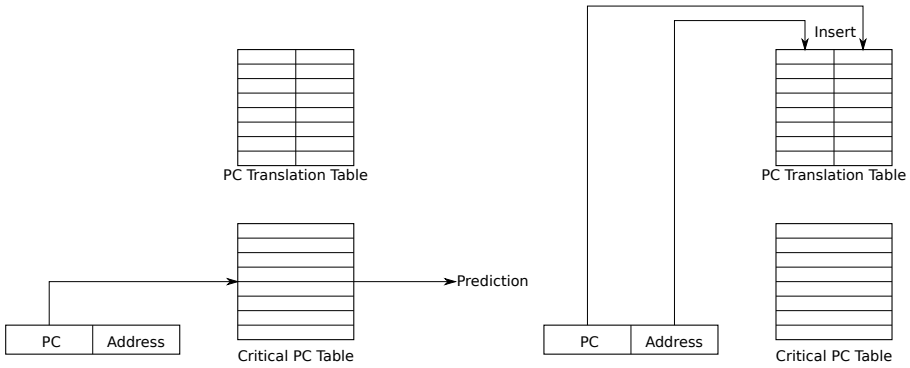
```
for(i=0; i<N; i++){
  res = foo();
  A[i+1] = A[i] + res;
  moo();
}
```

(b)

Figure 4.2: Address based checkpointing sometimes has trouble identifying dependences. Dependences through $p->count$ and A are not predictable through address.

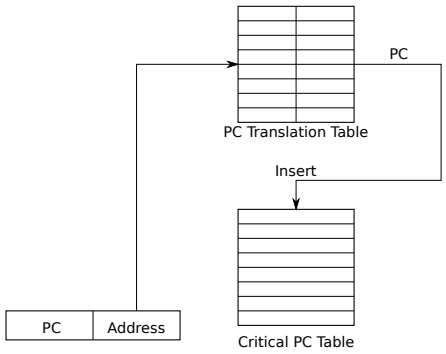
A table based mechanism is used for Program Counter based dependence prediction, similar to that used for address based prediction. The only change is that when a violation occurs, the Program Counter of the instruction performing the exposed load that led to the violation is inserted into the *Critical PC Table*, rather than the load address. For each instruction, a prediction can be obtained for checking whether the

Program Counter is contained in the criticality table[1]. If the PC is contained in this table, a violation is predicted.



(a) To make a prediction the PC is checked against the Critical PC Table.

(b) On an exposed load the PC and address are inserted in the PC Translation Table.



(c) On a violation, the PC corresponding to the address is inserted into the Critical PC Table.

Figure 4.3: PC Predictor in action.

Predicting dependences through the Program Counter does present one complication. When a dependence violation occurs, the program counter of the violating load is not directly available. The address of the violating store (and hence the violating load) is known. This means that this scheme requires some way of associating the address of an exposed load with the Program Counter of the instruction performing it. This can be done through another table, which we refer to as the *PC Translation Table*. When an

[1] Only load instructions can be critical, so to save power the table needs to be checked only for these.

exposed load occurs, the load address and the Program Counter are inserted into this table, shown in Figure 4.3b. It should be noted that this table is not on the critical path. The Program Counter corresponding to an address is *only* required when a violation occurs and the PC needs to be inserted into the Critical PC Table (4.3c). When making a prediction, only the Critical PC Table needs to be accessed, as shown in Figure 4.3a. Therefore some latency can be tolerated when accessing the PC Translation Table. The size required for this table depends on how much data is speculatively read by a task.

The effects on predictor performance of both the Critical PC Table and the PC Translation Table are discussed in Section 6.1.2.

4.1.3.3 Hybrid Prediction

Any system that runs a variety of workloads will encounter dependences that are predicted well through one kind of predictor but not the other. In fact, the same program may show both kinds of behaviour. The obvious solution is to use hybrid predictors, which employ both techniques.

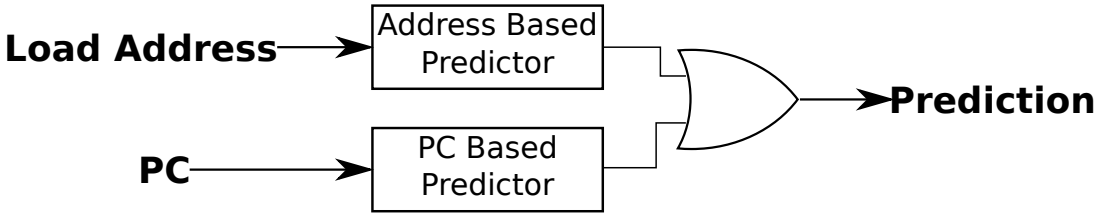


Figure 4.4: A hybrid dependence predictor where the outputs of the predictors are ORed.

The simplest way of achieving this is to have *both* address and program counter based predictors, in which case a positive prediction is returned if either predicts a dependence. That is, the outputs of the predictors are *OR*ed. This is shown in Figure 4.4. To build a more conservative predictor, the outputs can be *AND*ed.

A more sophisticated hybrid predictor can also be constructed. The presence of dependences on some instructions can be identified better through load addresses, and in others by the instruction address. This points to a construction where a program

counter based meta-predictor decides on which table to use for a particular instruction. Such a predictor is shown in Figure 4.5. The disadvantage of this predictor is that it is more complex and slower to train.

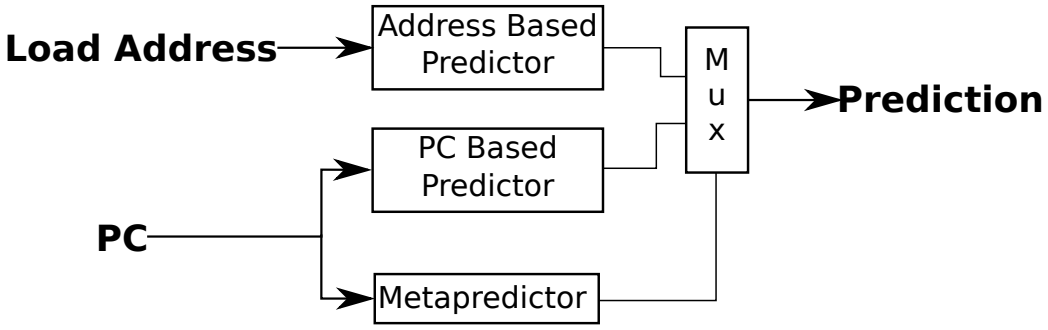


Figure 4.5: A hybrid predictor. Based on the Program Counter, a meta-predictor selects the prediction to use.

As with conventional hybrid branch predictors (24), the meta-predictor is updated only when the predictors disagree, while the other components are always updated. However, the meta-predictor is trained in a different manner than it traditionally is for branch prediction. It was discussed in Section 4.1.3 that predicting violating loads correctly is more important than total accuracy. Keeping this in mind, the meta-predictor is trained such that when the predictors disagree on a load that actually violates, the meta-predictor is saturated in favour of the correct predictor. The counter is only incremented/decremented in the opposite case.

Meta-predictors have been well studied in the branch prediction domain. They have been used to select (11; 24) or combine (36) the outputs of individual predictors.

4.2 Dealing with Resource Constraints

In a realistic checkpointing system, there will be some limit on the number of checkpoints it is possible to place per task. Further, there will be some overhead associated with placing checkpoints. This may be in latency or in additional speculative buffer state required, as discussed earlier in Section 3.2.2. In the presence of such limits

and costs, it may not be profitable to place a checkpoint on every positive prediction. Hence, the insertion policy takes resource constraints into account.

4.3 Checkpoint Insertion Policy

Once a data dependence violation is predicted, the system has to decide whether or not to insert a checkpoint. In the simplest case, we insert a checkpoint on every predicted dependence. This can lead to very small tasks, and if we assume even a small overhead for checkpointing, this does not remain efficient. The number of checkpoints available may also be limited.

We construct a hybrid heuristic which takes these issues into account and becomes more cautious about inserting a checkpoint as the number of checkpoints for a task increases. The policy chosen for checkpointing is the following:

$$\begin{aligned} \mathit{DoCheckpoint} \quad = \quad & (\text{Dependence Predicted}) \wedge (CP < CP_{max}) \wedge \\ & \left(\text{Size of Task} > \frac{C}{CP_{max}} \cdot (\text{CP} + 1) \right) \end{aligned}$$

Where CP_{max} is the maximum number of checkpoints allowed, and CP is the number of checkpoints already placed. *Size of Task* is measured in the number of instructions from the start of the task, or the latest checkpoint if a checkpoint has already been placed. Here, the constant C is the threshold for task size when choosing to place the last available checkpoint (when $CP = CP_{max} - 1$). The effect of the heuristic is to have a low threshold for task size when a large number of checkpoints are available, and increase it to the maximum value C when there is only one more checkpoint available. The value for C is chosen experimentally to be 100. This heuristic is evaluated in Section 6.2.

4.4 Microarchitectural Interactions

Beyond the basic architectural support for checkpoints discussed in the previous chapter, checkpoint insertion schemes have their own interactions with the microarchitec-

ture.

Even though we treat checkpoints as speculative tasks, the existing task spawning mechanism may not be sufficient. When a checkpoint is inserted, it is important that loads and stores on either side of the checkpoint boundary are marked with the correct task ID in the versioned cache. This issue has been discussed briefly in Section 3.1.1.

When (non-checkpoint) spawns are performed based on spawn instructions, the spawn can be identified when the instruction is decoded, so tracking memory accesses correctly is not problematic. Runtime dependence prediction and checkpoint insertion is not always as straightforward, as explained below.

The stride case is simple. Since the decision to insert a checkpoint is based on instruction counts, the instruction to be checkpointed can be known even before it is decoded.

For PC based checkpoints, if we assume a low latency prediction (reasonable based on the small Critical PC Table used), this is straightforward as well. Once again, we can have a prediction even before decode, since the prediction can be made when the instruction is fetched.

Address based prediction is far more problematic. A prediction cannot be made until the load address is available. The address may be computed late in the pipeline, and out-of-order processors will already have reordered instructions. It is necessary for *correctness* that a load which is in the parent thread not be marked as being in the child thread. If this does happen, it is possible that the checkpoint will receive a violation that *should* go to its parent, resulting in execution not being rewound as far as it should be and possible incorrect state. The converse is a performance issue, but does not make execution incorrect. This can be seen by considering the case where a load belonging to the child thread gets marked with the parent's task ID. If this location then receives a violation, it is the parent task that gets restarted. This will result in correct execution but with unnecessary re-execution.

This means, for the sake of correctness, the system either has to ensure that the load in question does not get issued before any loads that it follows, or alternately has to correct ID's if this does occur. Another way of dealing with the problem is to conservatively place the checkpoint before the instruction that is ready to retire when

the load to be checkpointed is identified. This will mean that some of the loads/stores that belong to the checkpoint will have been issued with the parent's ID. As mentioned above, this has performance repercussions but is not a correctness issue. It also means that if the checkpoint *is* restarted by a violation through the load that was predicted to squash, execution is rewound further back than is necessary.

We see that stride and PC based prediction are easier to implement, and PC based prediction can deal with latency in obtaining predictions and hence can allow the use of larger tables or more complicated prediction techniques.

Chapter 5

Evaluation Methodology

This chapter discusses the manner in which the checkpointing techniques and dependence predictors presented in earlier chapters are evaluated.

The first issue discussed is that of evaluation metrics. To perform a quantitative evaluation, meaningful measurements must be made and compared. Section 5.1 goes through the metrics that are used for the evaluation and justifies their selection. Then, Section 5.2 looks at the simulator used for obtaining these measurements. The configuration parameters of the system modeled are also given. Finally, in Section 5.3, the benchmarks used for the evaluation are discussed.

5.1 Evaluation Metrics

5.1.1 Evaluating Dependence Predictors

In order to compare the performance of the dependence predictors evaluated, we must look at metrics that reflect the usefulness of these predictors when used to place checkpoints. This section discusses the metrics used for evaluation.

A confusion matrix for predictions and outcomes is shown in Table 5.1. If a load is predicted as a dependence, and the prediction is correct, it is referred to as a *True Positive* (TP). If it predicted as a dependence and is actually not a dependence, this is a *False Positive* (FP). Similar terms are used for negative predictions as shown in the table.

Prediction	**Outcome**	
	Dependence	No Dependence
Dependence	*True Positive*	*False Positive*
No Dependence	*False Negative*	*True Negative*

Table 5.1: Prediction Outcomes.

Dependence prediction is a binary classification problem. The traditional measures used for such predictors are Accuracy, Sensitivity and Specificity.

The *Accuracy* of a predictor is simply the proportion of correct predictions made out of the total number of predictions.

$$Accuracy = \frac{True\,Positives + True\,Negatives}{True\,Positives + False\,Positives + True\,Negatives + False\,Negatives}$$

Sensitivity, also known as Recall, refers to the proportion of dependences that are correctly predicted.

$$Sensitivity = \frac{True\,Positives}{True\,Positives + False\,Negatives}$$

Specificity is the analogous measure for the negative case, i.e. the proportion of no-dependance outcomes that are correctly predicted.

$$Specificity = \frac{True\,Negatives}{False\,Positives + True\,Negatives}$$

There is one additional measure we look at. The *Precision*, also called the *Positive Predictive Value* (PPV), of a predictor is a measure of the accuracy of positive predictions.

$$Precision = \frac{True\,Positives}{True\,Positives + False\,Positives}$$

In the context of checkpointing, if we use a certain predictor directly for placing checkpoints, the Sensitivity of the predictor tells us what proportion of violating loads are checkpointed, and the Precision tells us what proportion of checkpoints placed was actually needed. The Accuracy and Specificity of the predictor are not as useful to directly reflect the behaviour of a predictor when placing checkpoints. Therefore, for the purposes of evaluating dependence predictors, the metrics we concentrate on are Sensitivity and Precision.

Another commonly used measure employed is the *F-measure*. This is the weighted harmonic mean between the Sensitivity and the Positive Predictive Value.

$$F_\beta = \frac{(1+\beta^2)(Precision \cdot Sensitivity)}{\beta^2 \cdot Precision + Sensitivity}$$

Or,

$$F_\beta = \frac{(1+\beta^2) \cdot True\,Positives}{((1+\beta^2)True\,Positives + \beta^2 False\,Negatives + False\,Positives)}$$

The F-measure for a perfect predictor is 1, and the worst possible value is 0. The F_1-measure, where $\beta = 1$, evenly weights Sensitivity and Precision. Higher values for β weight the Sensitivity higher, and values lower than 1 weight the Precision higher. The advantage of using the F_β-measure is that it reduces the performance of a predictor to one easily comparable number. The disadvantage that follows is that the comparison is only meaningful if the weighting chosen is reflective of the comparative advantage of a useful checkpoint versus the cost of a wasted checkpoint. This depends on system characteristics such as the overhead of placing a checkpoint and the re-execution saved when a checkpoint is correctly saved. To keep the evaluation as general as possible, over-reliance on this single measure is avoided.

Benchmark	Known Prediction Outcomes (%)
bzip2	78
crafty	71
gap	43
gzip	91
mcf	83
parser	63
twolf	95
vortex	94
vpr	64

Table 5.2: Percentage of Outcomes Known.

5.1.1.1 Measuring Dependence Predictor Performance

For the purpose of evaluating a dependence predictor, the outcome (as in Table 5.1) has to be measured for each prediction made. However, not all predictions have an associated outcome. When a task receives a violation and is restarted or killed, apart from the address receiving the violation, the outcomes for the loads performed by the thread are not known. The outcomes known are those for violating loads and for all loads within tasks that commit.

All results reported for dependence predictors in Chapter 6 necessarily take into account only known outcomes. Table 5.2 shows, for each benchmark, the fraction of predictions made that have a known outcome. This fraction is fairly high, and is below 50% for only one benchmark (*gap*). This means that when evaluating predictors, we are able to take into account a large portion of the predictions made.

5.1.2 Evaluating Checkpointing Schemes

Beyond the effectiveness of dependence predictors, the checkpoint insertion schemes also need to be evaluated in terms of the savings they provide. Once again, an attempt

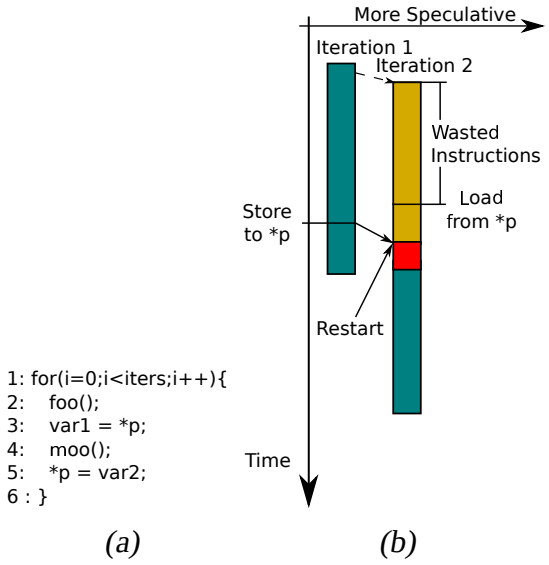


Figure 5.1: The instructions before the load from *p are not dependent, and are wastefully re-executed. All the instructions from the start of the task to the one immediately preceding the load are *wasted instructions*.

has been made to keep the evaluation as general as possible.

The purpose of checkpointing is to reduce wasteful re-execution. A direct measure of wasteful re-execution is the number of unnecessarily squashed instructions. When a task is squashed, the violated load and every instruction after that has to be re-executed. However, in practice, there may be instructions between the start of the task and the offending load. These instructions do not have to be re-executed to maintain correctness, and re-execution only occurs as a consequence of where task boundaries are placed. Here these instuctions are referred to as *wasted instructions*. An example is shown in Figure 5.1, where all the instructions before the load from *p in the speculative task are wasted instructions.

As with dependence prediction, the evaluation must reflect both the savings and the potential cost in overhead. Where savings in re-executed instructions are important, in cases where there is a cost associated with placing a checkpoint, it is important to take that into account. So, alongside the total savings in wasted instructions, checkpointing is also evaluated in wasted instructions saved per benchmark.

The checkpointing schemes are also evaluated using the traditional metrics of exe-

cution time, power and energy.

5.2 Simulator

We conduct our experiments using the SESC simulator (33). SESC can model different processor architectures, such as single processors, chip multi-processors and processors-in-memory. It models a full out-of-order pipeline with branch prediction, caches, buses, and every other component of a modern processor necessary for accurate simulation. The simulator leverages the MINT emulator (47) to generate instruction objects, which the event driven SESC simulator then uses for timing simulation.

The SESC simulator has been extended to include the checkpointing support described in Chapter 3 and the dependence predictors described in Chapter 4.

The main microarchitectural features of the baseline system are listed in Table 5.3. The system we simulate is a multicore with 4 processors, where each processor is 4-issue out-of-order superscalar. For the TLS protocol we assume out-of-order spawning (34). The latencies of all the caches were computed based on CACTI (43). The power consumption numbers are extracted using CACTI and Wattch (2).

5.3 Benchmarks

We use the integer programs from the SPEC CPU 2000 benchmark suite running the Reference data set. We use the entire suite except *eon*, *gcc* and *perlbmk*, which failed to compile in our infrastructure. The TLS binaries were obtained with the POSH infrastructure (18). A subset of the SPEC CPU 2006 benchmarks is also used. These are *astar*, *bzip2*, *mcf* and *sphinx3*, running the Training data set. The running times for these benchmarks is much longer than for the SPEC 2000 programs, so the Training data set is used to keep simulation time feasible. This subset of SPEC 2006 was chosen because it shows good TLS potential and is not trivial to parallelise (29). For these benchmarks, high coverage loops have been selected for speculation. *bzip2* and *mcf* show behaviour very similar to their SPEC2000 counterparts, hence they are not discussed separately in the Evaluation chapter.

Parameter	TLS (4 cores)
Frequency	5GHz
Fetch/Issue/Retire Width	4, 4, 5
L1 ICache	16KB, 4-way, 2 cycles
L1 DCache	16KB, 4-way, 3 cycles
L2 Cache	1MB, 8-way, 10 cycles
Main Memory	500 cycles
I-Window/ROB	40, 100
Branch Predictor	16Kbit Hybrid
BTB/RAS	1K entries, 2-way, 32 entries
Cycles from Violation to Kill/Restart	12
Cycles to Spawn	12

Table 5.3: Architectural parameters used.

In order to compare sequential, TLS and checkpointed execution, we need to make sure that the same code segments are executed in each case. Traditionally, this is ensured by executing a given number of instructions. For speculative systems, however, the instruction count may differ depending on the amount of misspeculation. For this reason, we place *simulation marks* across the code regions we wish to simulate and make sure that evaluations are based on the same code segments. This is also necessary because the sequential and TLS binaries are different, due to re-arrangements of the code by POSH. After skipping the initialisation phase for each benchmark, enough simulation marks are simulated so that the corresponding sequential application graduates more than 750 million instructions.

Chapter 6

Results and Evaluation

In this chapter, a quantitave analysis is performed of the techniques described earlier in the thesis. First, in Section 6.1, the various dependence predictors described in Section 4.1.3 are evaluated and compared, and various configuration options considered. Then, Section 6.2 evaluates the performance of checkpointing policies and explores parameter selection for these. Section 6.3 looks at the effects of the hardware mechanisms described in Section 3.2, including memory system optimisations and selective kills and restarts. Finally, Section 6.4 compares checkpointing with synchronising loads that are predicted to be dependent

6.1 Dependence Prediction

Various dependence prediction mechanisms were described in Section 4.1.3. Here, the performance of these address based, Program Counter based and hybrid predictors is evaluated. The effects of structure sizes are analysed and in Section 6.1.4 the performance of address based, Program Counter based and hybrid predictors is compared. For the evaluation of the predictors, a four processor system running Thread Level Speculative code is simulated. No checkpointing is performed for this evaluation. This allows easier comparison of predictors, since predictions do not modify execution and no secondary effects are introduced, hence each predictor sees exactly the same memory accesses and dependences.

6.1.1 Address Based Prediction

The first evaluation performed is of the performance of an address based predictor as described in Section 4.1.3.1. The plots in Figure 6.1 show the Sensitivity of the predictor along with the Precision as the size of the Critical Address Buffer is varied from 2 to 128. The Sensitivity increases along the x-axis, while the Precision increases along the y-axis. Desirable predictor performance would be in the top right corner of the graph, with Precision and Sensitivity being close to one. As the results in this chapter show, in practice, tuning predictors to be more aggressive leads to higher Sensitivity, but lower Precision. The choice of these metrics to evaluate squash predictors has been discussed in Section 5.1. The address based predictor is evaluated for Random, First-In-First-Out (FIFO) and Least Recently Used (LRU) replacement policies. Each line in the plots corresponds to a replacement policy. The datapoints have been annotated with the number of entries in the Critical Address Buffer.

The results show that, as expected, larger buffer sizes result in higher Sensitivity, but at the cost of a larger number of false positives, reflected in lower Precision. What this means in terms of checkpointing is that if the predictor is used directly to place checkpoints, a larger buffer would result in better coverage. That is, a higher proportion of dependent loads would be checkpointed. The higher number of false positives, however, would result in a larger number of unnecessary checkpoints. It becomes important to avoid unnecessary checkpoints if there is a high cost to placing a checkpoint, or if the number of checkpoints allowed is limited.

It should be noted that the range of variation in Sensitivity is dramatically different for different benchmarks. *mcf* shows very little change in going from one entry upwards, while other benchmarks show strong improvement. Some of the benchmarks, for example *crafty* and *vpr* show improving Sensitivity right up to 64 entries, while others (*parser*, *gzip*, *vortex*) stop showing considerable improvement earlier. *twolf* is the only benchmark to show noticeable improvement up to 128 entries. The Precision also has very different ranges of variation for different benchmarks. Precision is extremely low for *gzip* (0 to 0.01) and *vortex* (0.01 to 0.04). These benchmarks do not see much of an effect from table size on the Precision. On the other hand, *bzip2*, *gap* and *mcf* show fairly high Precision of over 0.3 for many table sizes.

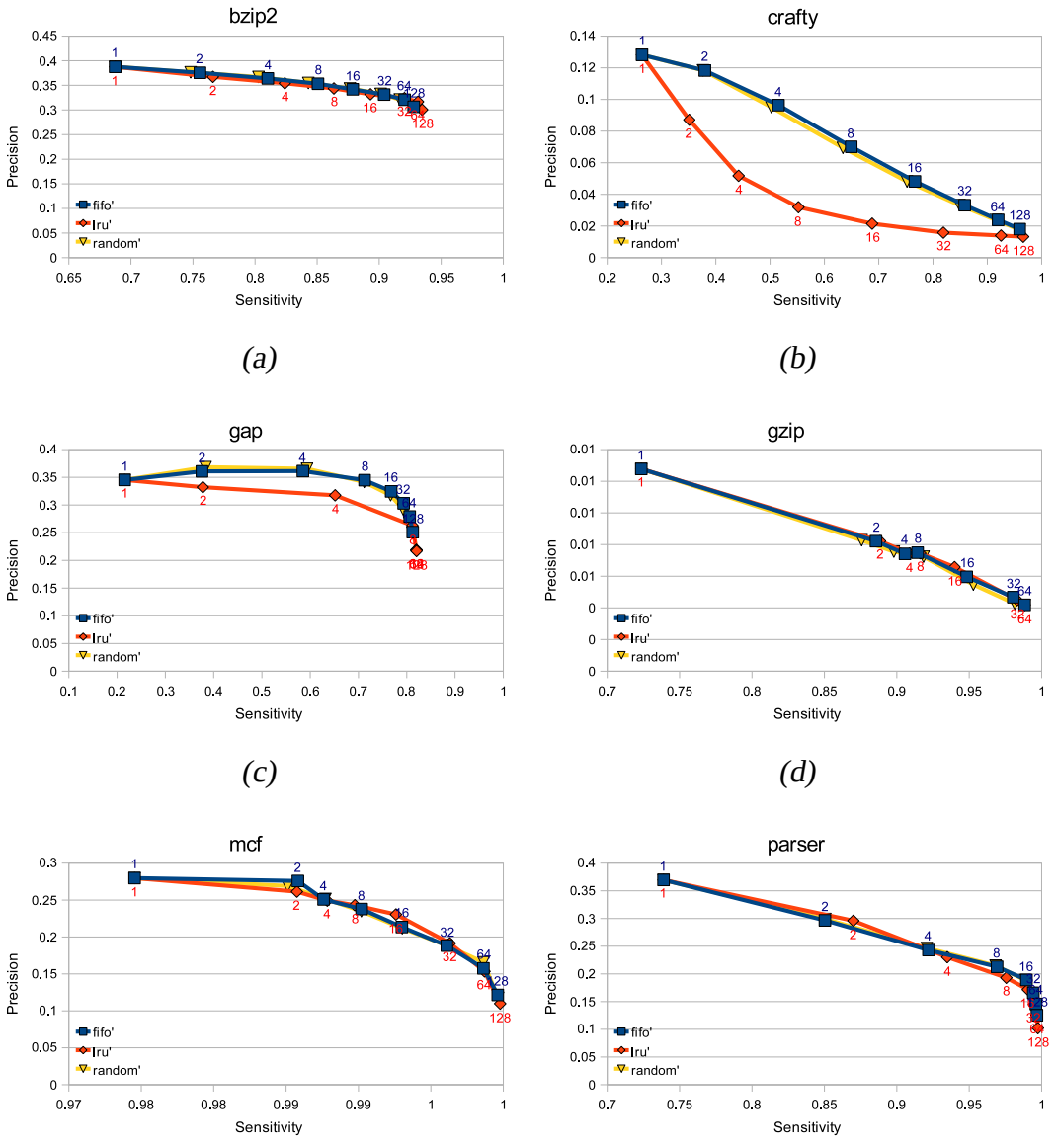


Figure 6.1: Sensitivity and Precision for address based dependence predictor with LRU, FIFO and Random replacement policies. (Continued on next page)

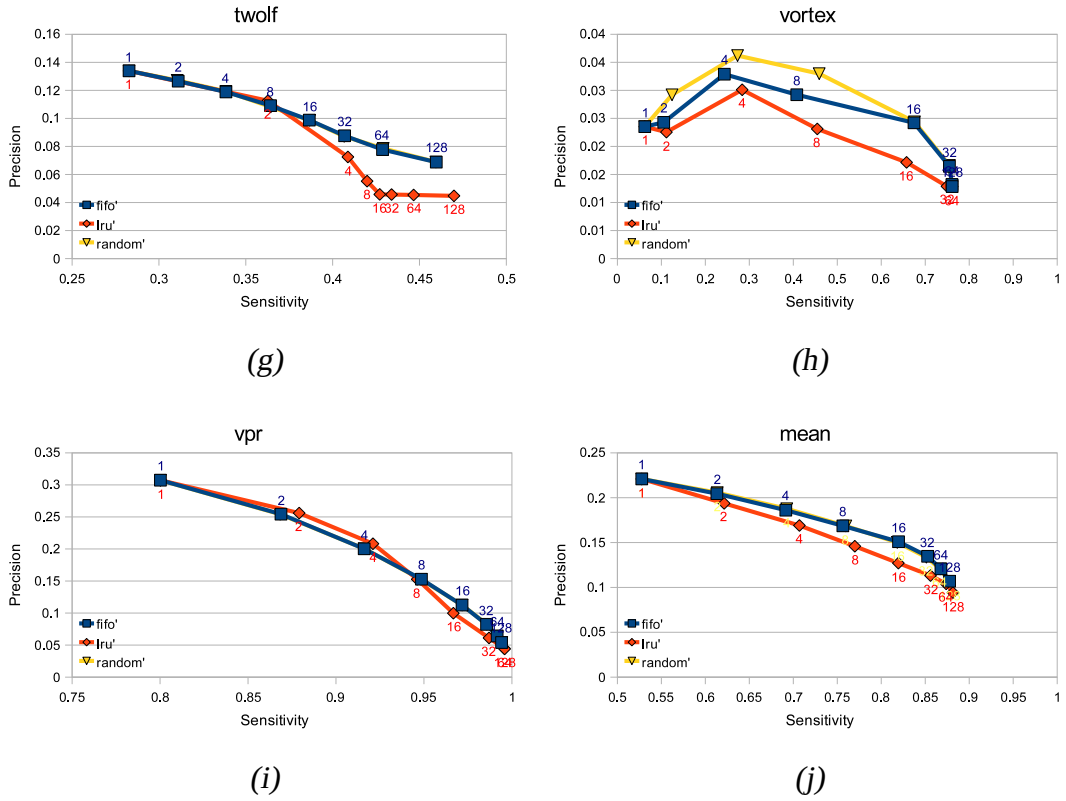


Figure 6.1: Continued: Sensitivity and Precision for address based dependence predictor with LRU, FIFO and Random replacement policies.

The comparison of replacement policies provides some interesting results. With only the exception of *crafty* and at some sizes *mcf*, LRU replacement provides greater Sensitivity for the same buffer size than FIFO or Random replacement. This is most pronounced at buffer sizes from 2 till 8. The fact that the effect lessens at large buffer sizes is explainable by the fact that at sizes above 8, buffer capacity is not the main limiting factor for Sensitivity. For a single entry buffer, the replacement policies are equivalent so no difference is observed.

Even though LRU replacement provides better Sensitivity for equal table size compared to the two other replacement policies, it can be seen that for most benchmarks, and on average for all benchmarks, the lines for FIFO and Random replacement lie above the line for LRU. This means that we can achieve a given Sensitivity with fewer false positives than is the case for LRU. This leads to the conclusion that, if we ig-

nore the power and area effects of buffer size, then for just the predictor performance tradeoffs, it is preferable to use FIFO or Random replacement than LRU. In fact, looking at the mean performance for all benchmarks for buffer sizes of 16 and 32, the difference in Sensitivity is negligible while the difference in Precision is much more significant. The only case where LRU replacement is preferable is when high Sensitivity is required from a comparatively small table of size 2, 4 or 8. Apart from this, the complexity of LRU replacement can be avoided and simpler replacement policies can be used for a better performing predictor.

The evaluation shows that the advantage of increasing the size of the Critical Address Table beyond 32 has little benefit in Sensitivity. This suggests that a 32 or 64 entry, FIFO replaced table is the best choice.

6.1.2 Program Counter Based Prediction

Figure 6.2 shows the performance of a Program Counter based predictor as described in Section 4.1.3.2. Similarly to the address based predictor, it is evaluated for Random, First-In-First-Out (FIFO) and Least Recently Used (LRU) replacement policies. For the results in Figure 6.2, an unlimited PC Translation Table is assumed.

The Program Counter based predictor behaves fairly similarly to the address based one discussed above as the Critical PC Buffer size and replacement policy are varied. Growing Sensitivity accompanied by falling Precision can be observed as the buffer size is increased. There is little benefit in Sensitivity shown above a buffer size of 16, and next to none above 32.

Apart from *crafty*, a buffer size of 8 sees all the benchmarks reach either 90% Sensitivity, or, in the case of *vortex* close to the maximum Sensitivity available.

Comparing the replacement policies shows that for the critical PC buffer, LRU replacement leads to Precision compared to Random and FIFO, and the Sensitivity is markedly better for only one benchmark - *vortex*. However, unlike the address based predictor discussed above, the difference between replacement policies is only significant for buffer sizes of 2 to 8. This still leads to the same conclusion: it is best to use a simpler Random or FIFO replacement policy since LRU adds complexity for no benefit.

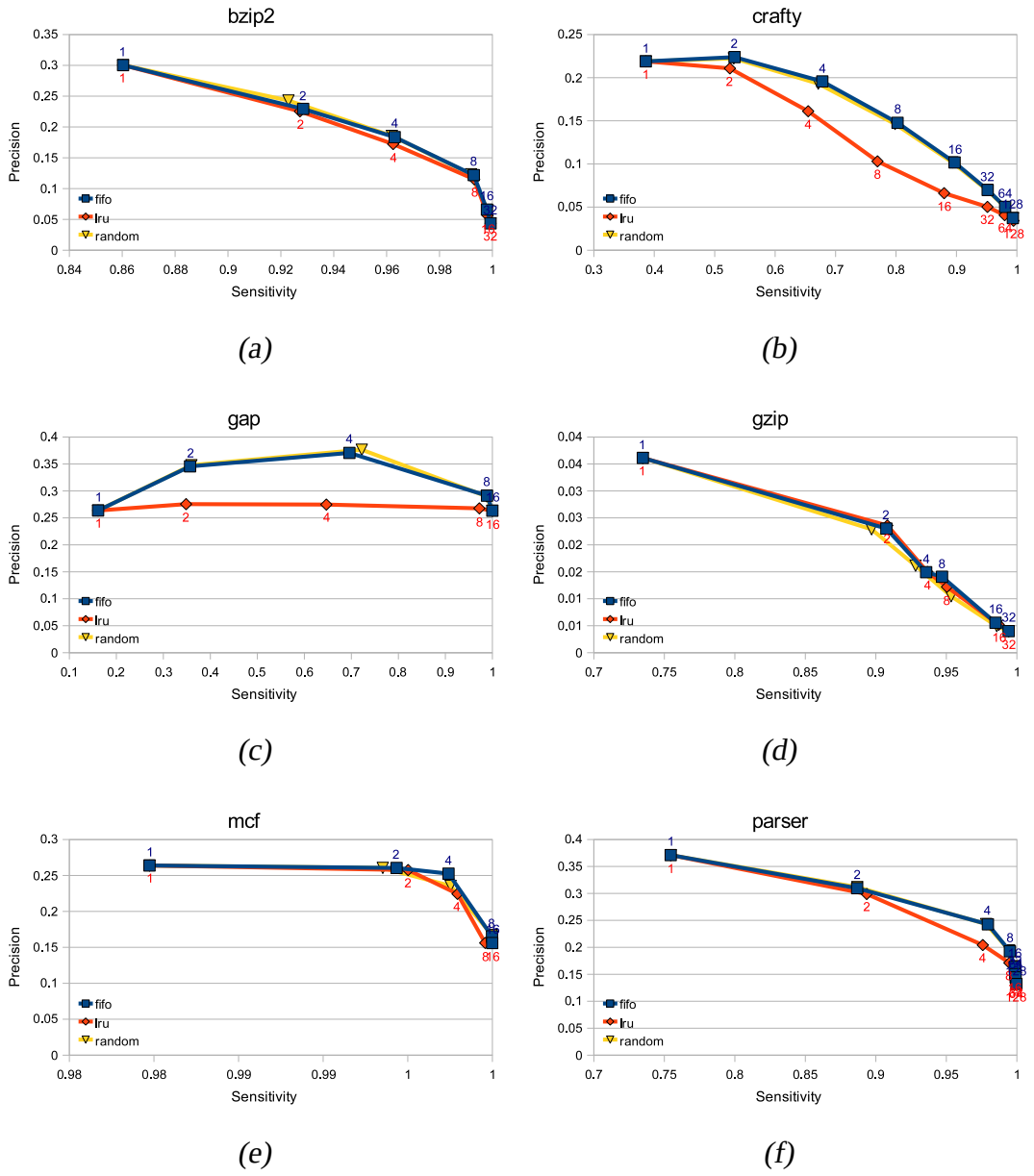


Figure 6.2: Sensitivity and Precision for Program Counter based table dependence predictor with LRU, FIFO and Random replacement policies. (Continued on next page)

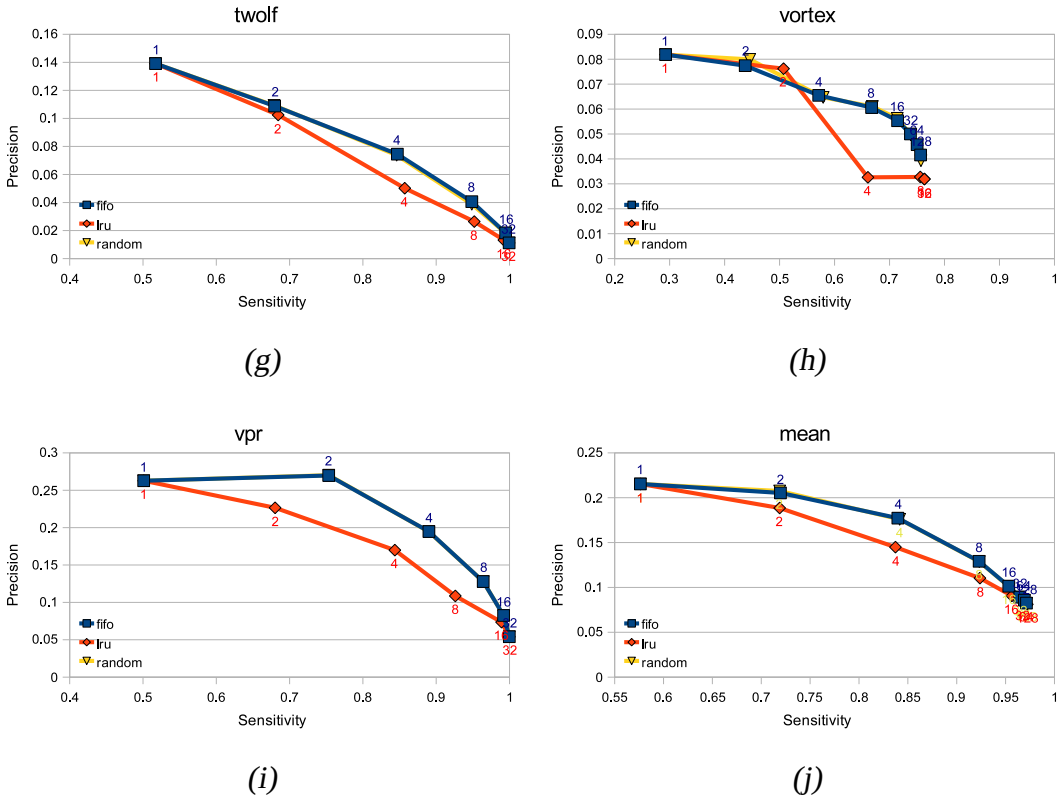


Figure 6.2: Continued: Sensitivity and Precision for Program Counter based table dependence predictor with LRU, FIFO and Random replacement policies.

In the results above, an infinite buffer was assumed for performing Address to PC translation on violations. Obviously this is not a realistic assumption. In Figure 6.3 the effects of varying the size of the PC Translation Table are shown. The sizes shown are 1, 2, 4, 8, 16, 32, 64, 128, 256 and Infinite. Each line corresponds to a different size of the Critical PC Buffer: 4,16 and 64.

It can be seen that translation table sizes of less than 16 achieve very low Sensitivity. For all the benchmarks, there is rapidly increasing Sensitivity to a certain table size, after which increase in the size becomes far less important. This size required to get close to the maximum benefit available varies from 16 for *bzip*, *gap* and *mcf* to 128 for *vortex*. *vortex* is also the only benchmark to show any improvement at all beyond a size of 128. The mean for all benchmarks shows rapid improvement in Sensitivity up to 64, some further increase to 128 and negligible improvements after that. These

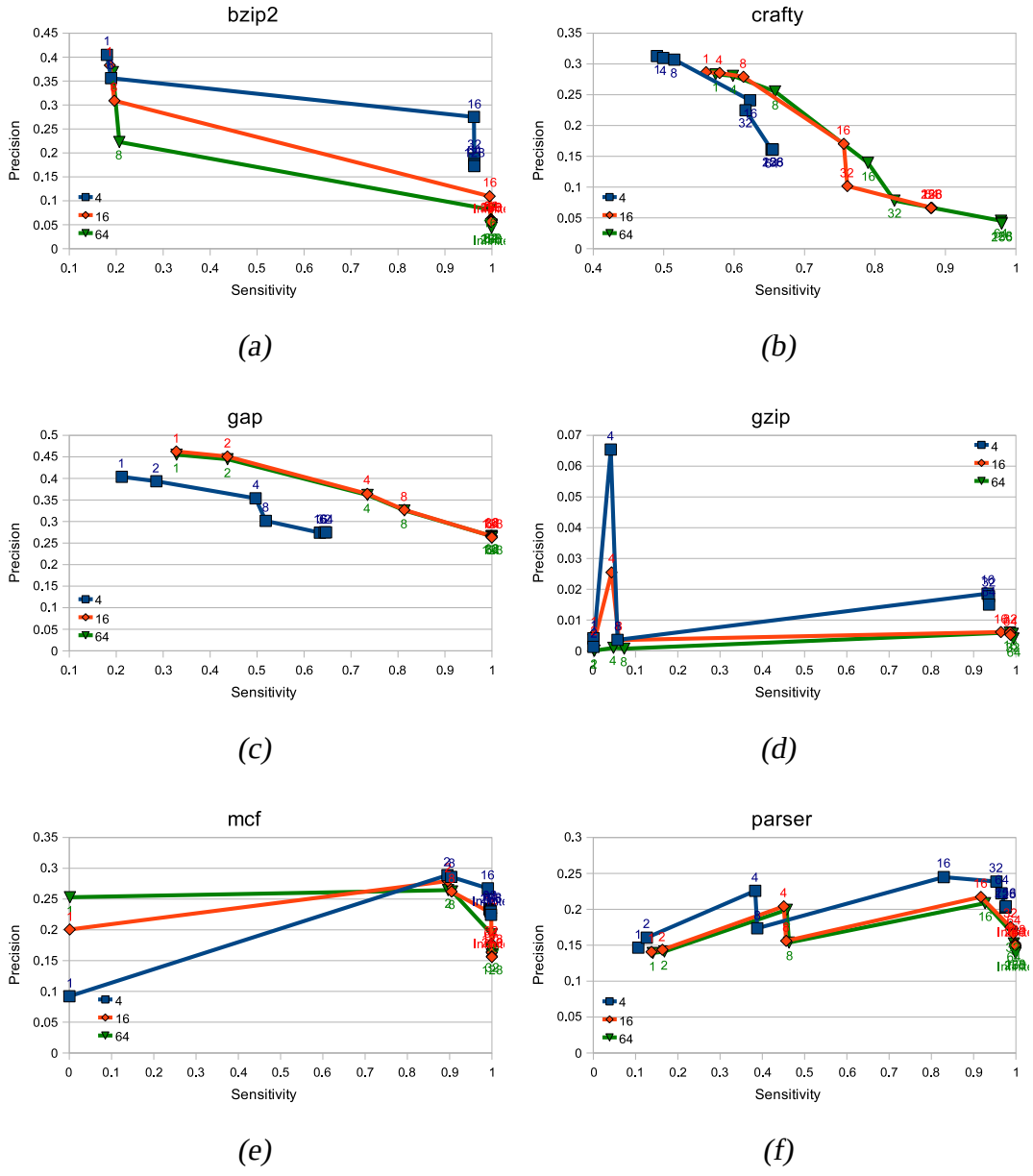


Figure 6.3: Sensitivity and Precision for PC based dependence predictor for various sizes of PC Translation Table. (Continued on next page)

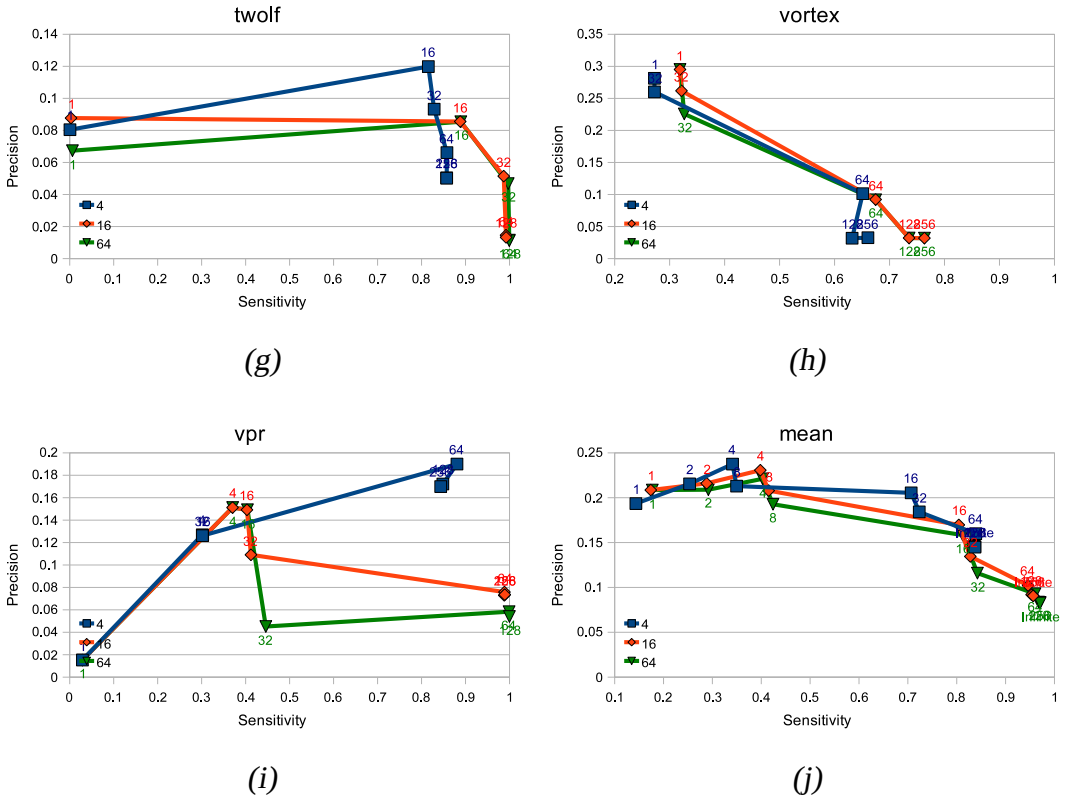


Figure 6.3: Continued: Sensitivity and Precision for PC based dependence predictor for various sizes of PC Translation Table.

results suggest that a size of 64 is sufficient to capture most dependences. This is true regardless of the size of the Critical PC Buffer.

It should be noted that like the PC Criticality Buffer, this is a fully associative table. If set associative or direct mapped storage is used, a larger size may be required. However, unlike the Criticality Buffer, the Translation Table is not on the critical path, so keeping the size small is not as much of a concern.

The evaluation for the Program Counter predictor shows that it reaches its best performance at a smaller sized criticality table compared to the address based predictor. There is little benefit beyond 8 to 16 entries in the PC Criticality Table. The PC Translation Table, however, requires 64 entries to show good performance. This suggests that the predictor should be configured with an 8 to 16 entry Criticality Table and a 64 entry Translation Table.

6.1.3 Hybrid Prediction

In this section the various hybrid predictors described in Section 4.1.3.3 are evaluated. In Figure 6.4, we evaluate the predictions obtained by taking the *OR* and *AND* of the address and Program Counter based tables discussed in the previous two sections. Buffer size for Critical Address and Critical PC buffers is varied from 1 to 128. The PC Translation table is fixed at 64 entries.

The *OR* hybrid predictor obtains an aggressive prediction, by prediction a dependence when either of the individual predictors returns a positive prediction. This leads to high Sensitivity, but a high false positive rate. The *AND* predictor is much more conservative, only predicting a dependence when both individual predictors agree on a positive prediction. This leads to a lower false positive rate, but obtains far less Sensitivity than the *OR* predictor.

Next, in Figure 6.5, we look at a hybrid bimodal predictor, using a direct mapped bimodal table to select which prediction to use. The lines in the figure are various sizes for this table. The critical address and PC buffers are fixed at 32. The points for each line show the number of bits for the counters in the table.

The results show that a very small metatable of 32 or 64 entries leads to both lower Sensitivity and Precision. This is because of aliasing between different PC values. Because the training method is biased towards maintaining Sensitivity (saturate on dependence, decrement on no-dependence), the aliasing has less of an effect on Sensitivity than on Precision, particularly for wider counters. Above 1k entries, the difference in predictor performance is negligible. Therefore, larger metatable sizes are not shown.

When we see the variation through the number of counter bits, we see steadily increasing Sensitivity. This points to behaviour where some dependences only occur rarely. Increasing the number of bits to a very large value would lead to behaviour where once a dependence is observed, the individual predictor returning a positive prediction is always selected. This would approximate the *OR* predictor.

Next, in Figure 6.6, we look at 128 entry and 1k entry tables, with 2 and 5 bits and observe the effect of varying the buffer size of the individual predictors from 1 to 128. As the results above suggested, a metatable with 5 bits controls the false positive rate

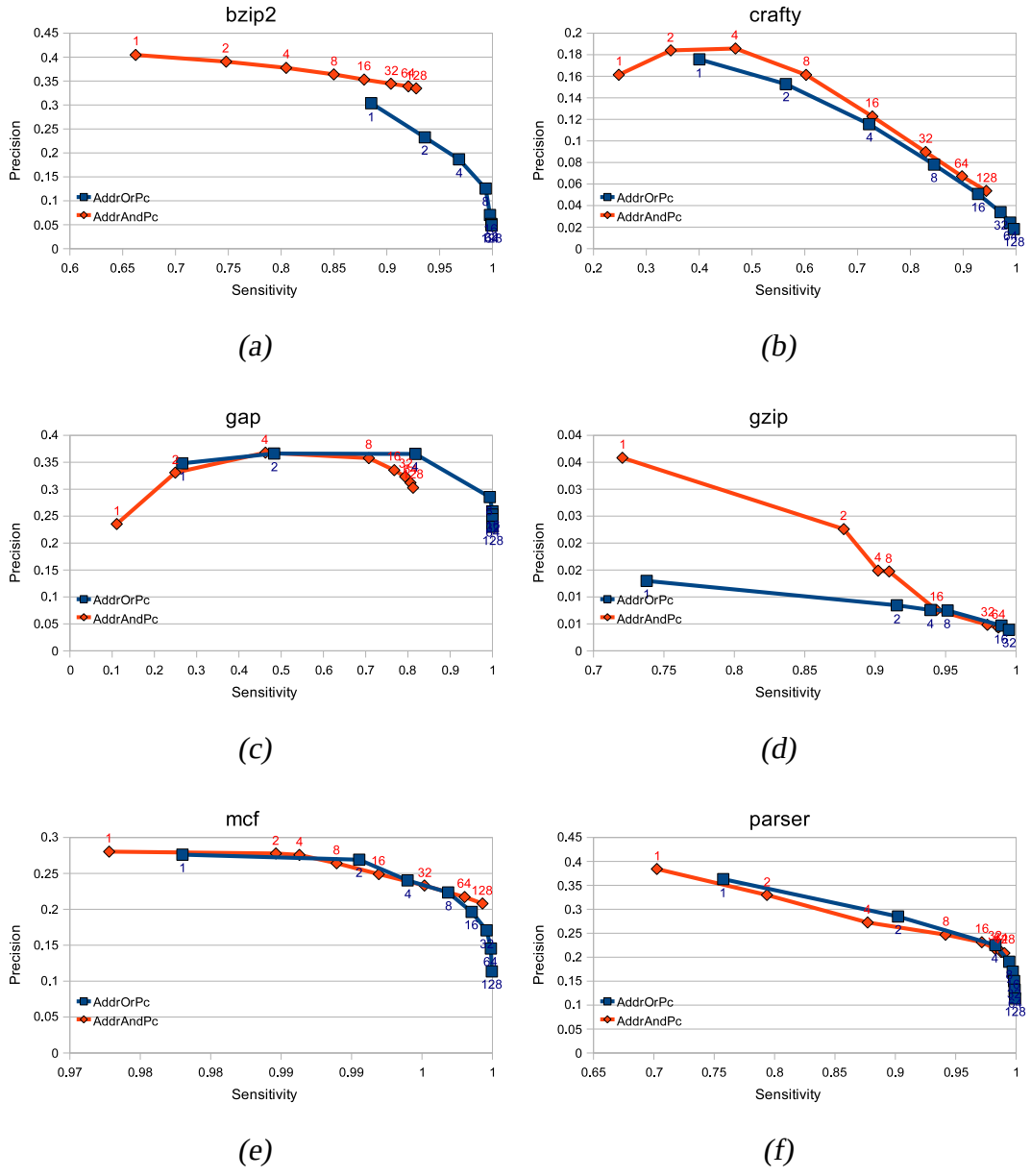


Figure 6.4: AND and OR hybrid predictors, varying buffer size for individual predictors. (Continued on next page)

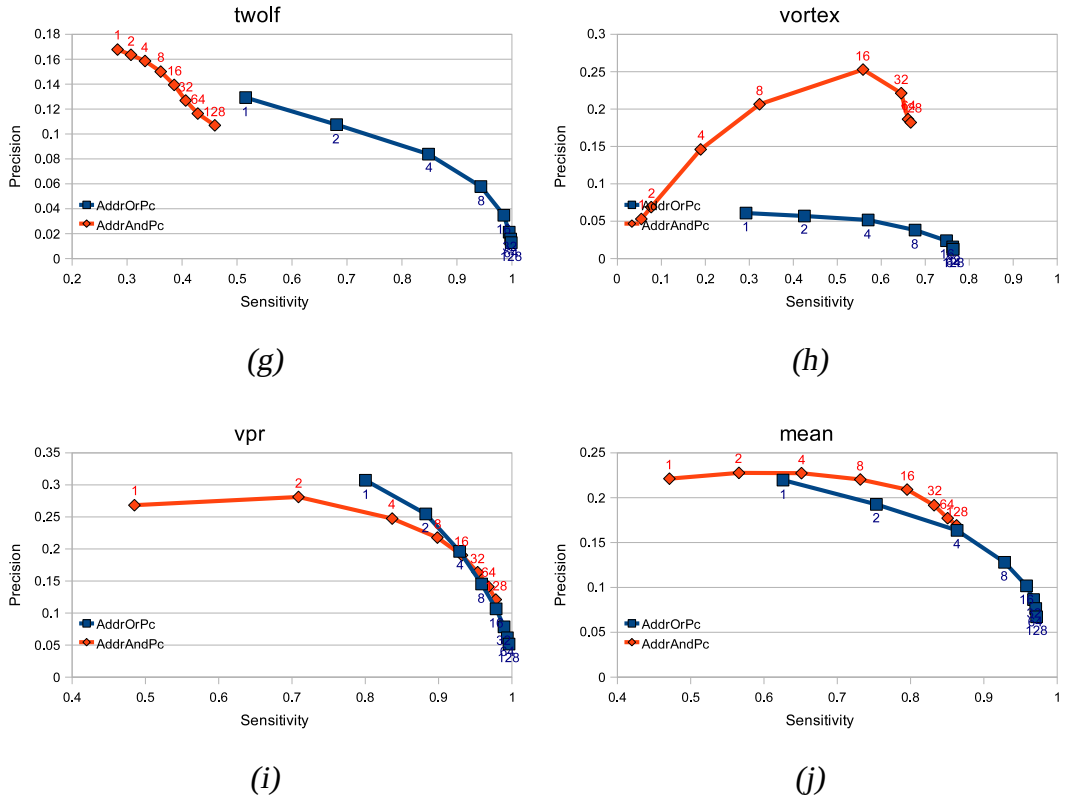
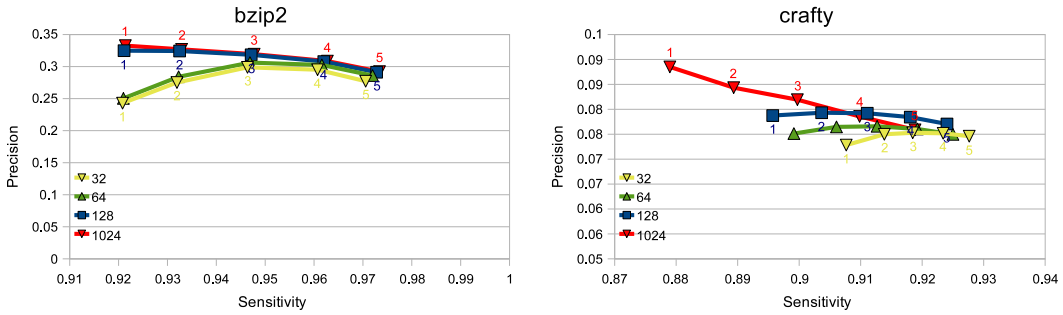


Figure 6.4: Continued: AND and OR hybrid predictors, varying buffer size for individual predictors.

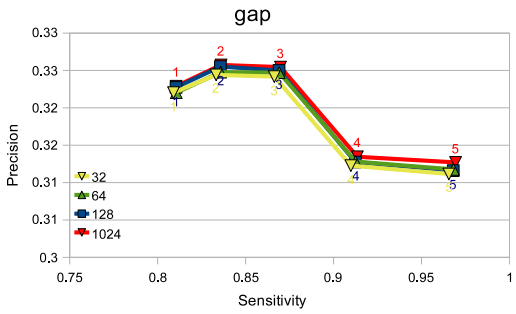
best while providing good Sensitivity as well.

One interesting thing to note from these results is that for both Sensitivity and the tradeoff between Sensitivity and Precision, the number of counter bits is much more important than the number of entries in the table (beyond very small values). This reflects the fact that many important dependences are infrequent, and to maintain Sensitivity, the predictor has to have long memory. The design related result that follows from this is that a comparatively small metatable is sufficient, but it is important to use a wide enough counter.

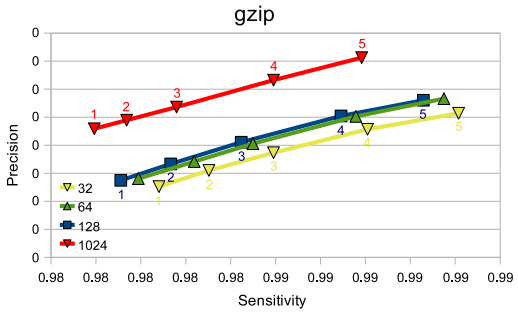


(a)

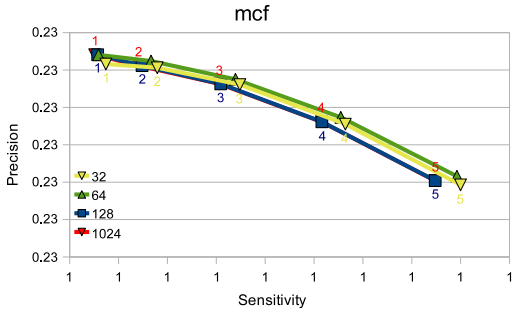
(b)



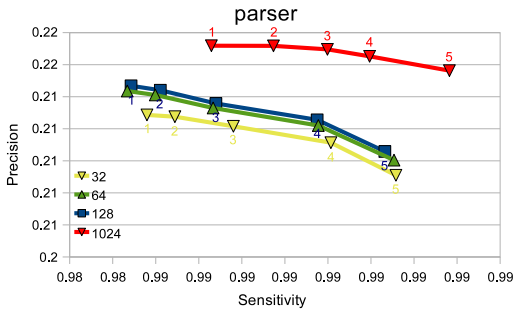
(c)



(d)



(e)



(f)

Figure 6.5: Hybrid bimodal predictor, varying table size and the number of counter bits. (Continued on next page)

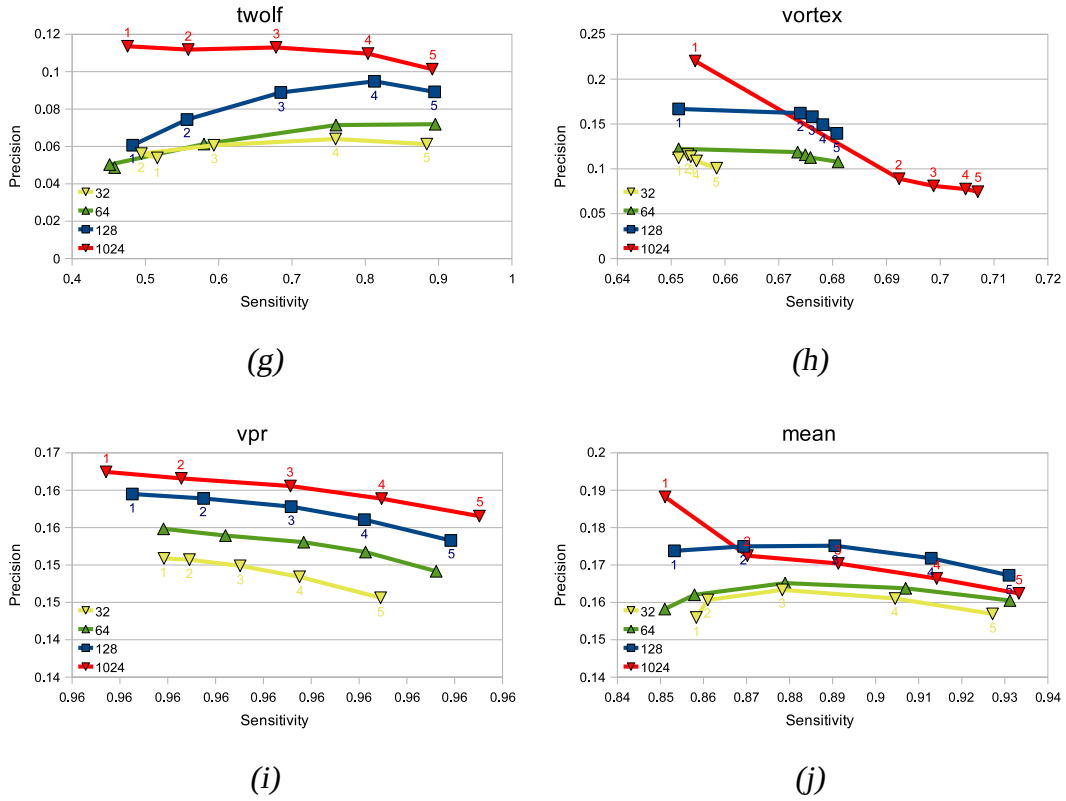


Figure 6.5: Continued: Hybrid bimodal predictor, varying table size and the number of counter bits.

6.1.4 Comparison of Predictors

Figure 6.7 shows a comparison between the performance of the predictors discussed so far. For all buffers, a First-In-First-Out (FIFO) replacement policy is used, and Critical Address and Critical PC Buffer sizes are varied from 1 to 128. For the hybrid bimodal predictor, a 128 entry, 5 bit metatable is employed. For all the PC predictors used, a 64 entry PC Translation Table is modeled.

It can be seen that for most benchmarks (with the notable exception of *vortex*), and on average for all benchmarks, the line for the hybrid bimodal predictor lies well above any of the other predictors. This means that we can achieve a given Sensitivity with fewer false positives. When we compare the simpler address and Program Counter based predictors, we observe higher Sensitivity for a given buffer size if we

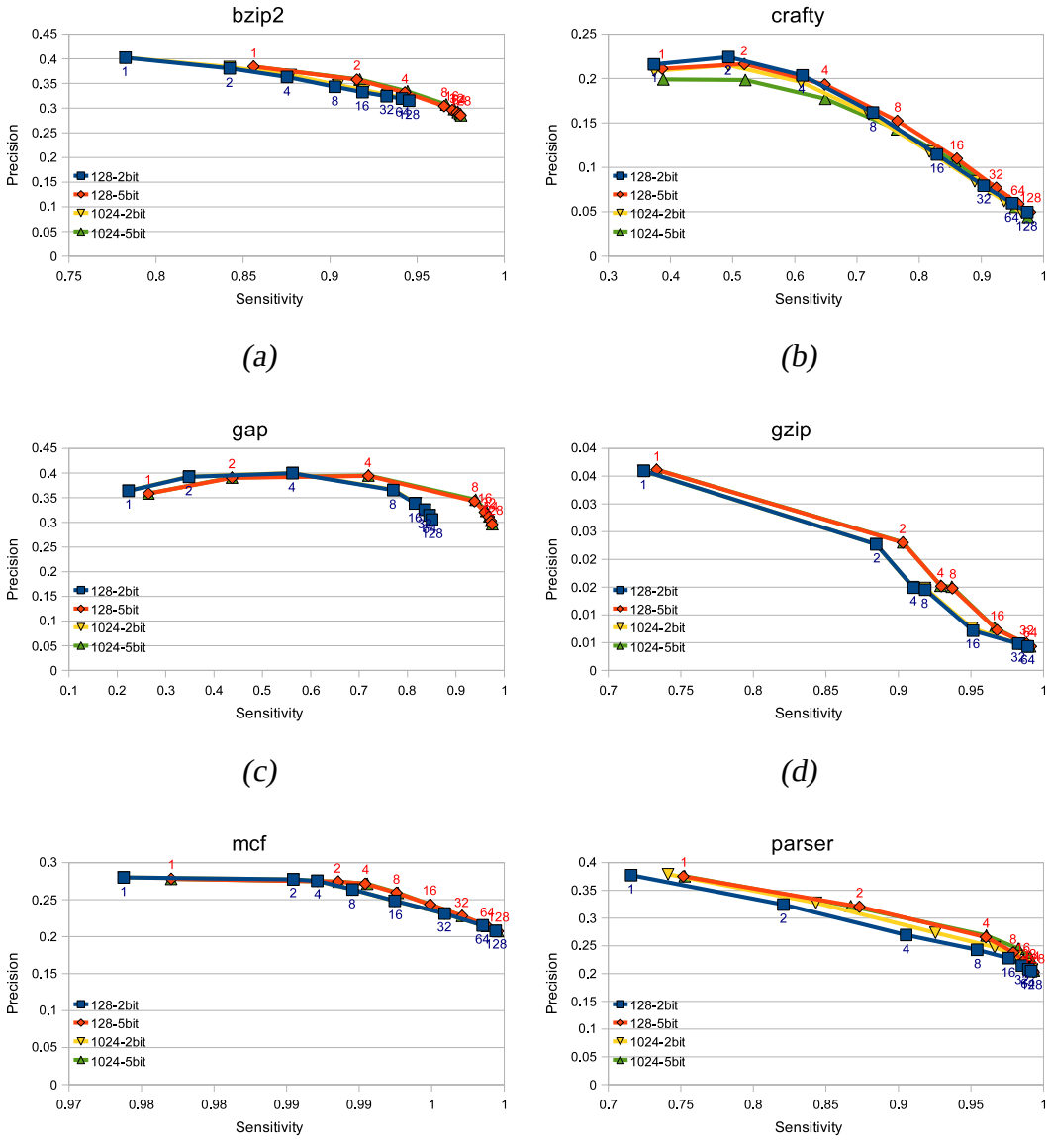


Figure 6.6: Hybrid bimodal predictor, varying buffer size for individual predictors. (Continued on next page)

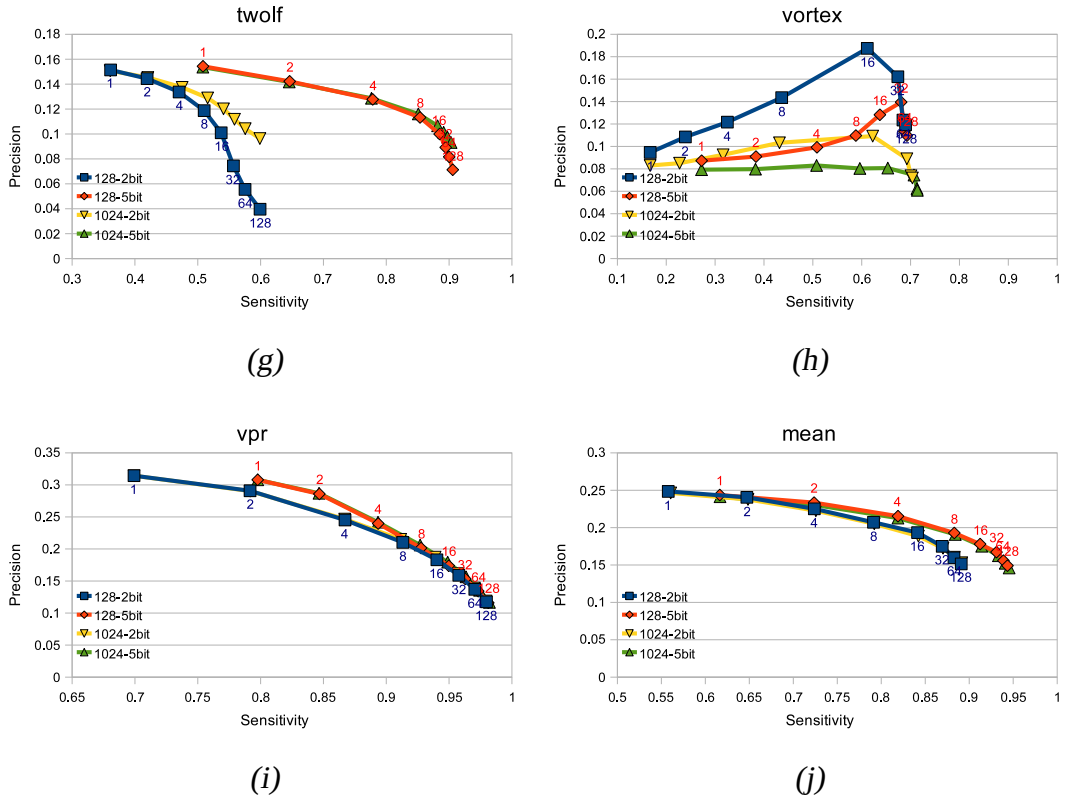


Figure 6.6: Continued: Hybrid bimodal predictor, varying buffer size for individual predictors.

use Program Counter based prediction. This is true for all the benchmarks. However, bzip2, and for some table sizes, gap and twolf show a better tradeoff between Sensitivity and true positive rates for an address based predictor. Of the predictors evaluated, the best Sensitivity is achieved by the aggressive OR predictor. This is at the cost of a high False Positive rate, particularly at larger buffer sizes. The more conservative AND predictor provides a better tradeoff, but still not as good as the hybrid bimodal predictor.

Table 6.1 shows the F_β values for the predictor types. The F_β measure has been discussed earlier in Section 5.1. Two sets of comparisons are shown. On the left, the Address predictor is configured with a Critical Address Buffer of size 8. For a fairer comparison, since the Program Counter predictor requires the PC Translation Table, it is configured with half the buffer size. The hybrid predictors, since they need

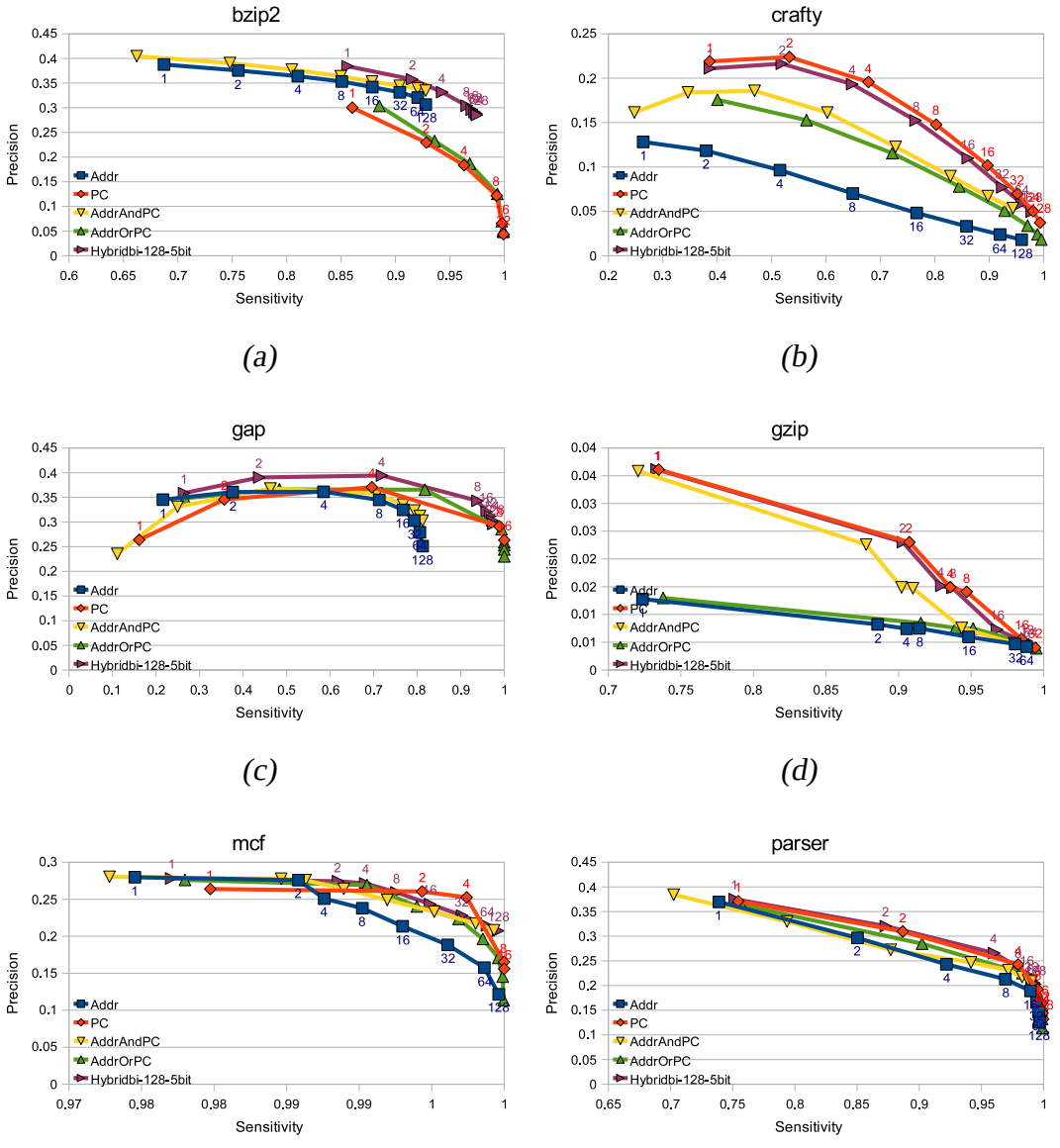


Figure 6.7: Comparing various predictors. (Continued on next page)

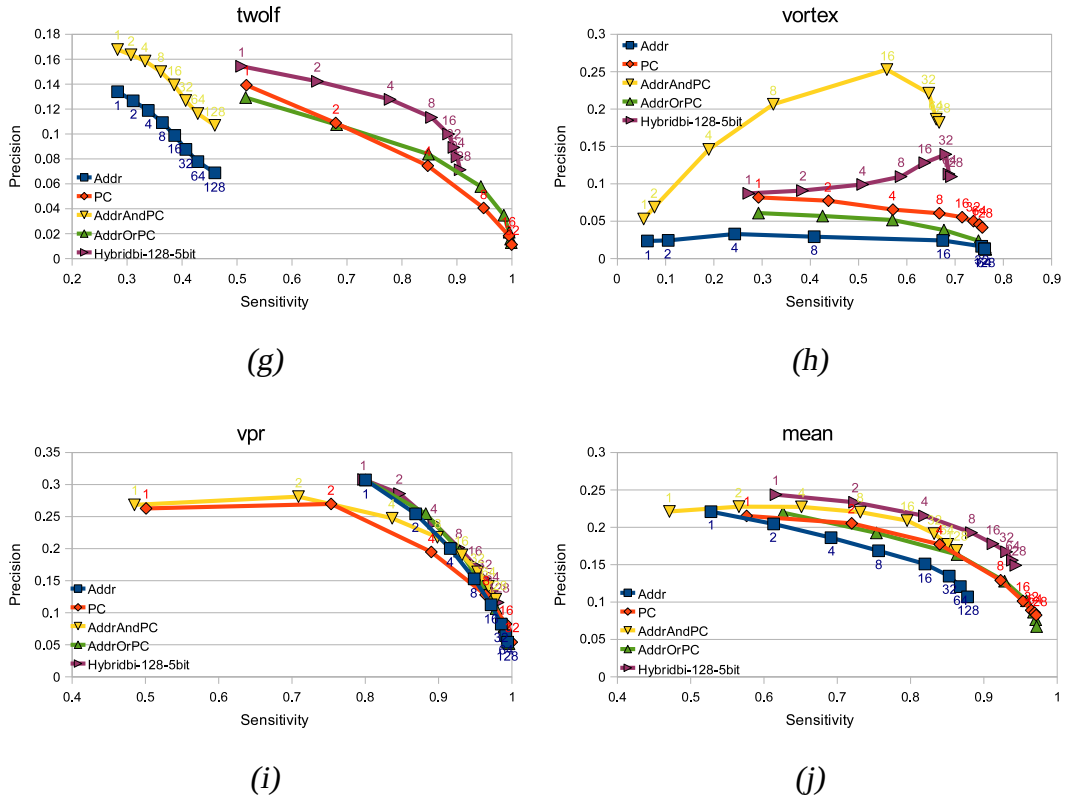


Figure 6.7: Continued: Comparing various predictors.

Predictor	F_1	F_2	F_3	F_7	Predictor	F_1	F_2	F_3	F_7
Address (8)	0.28	0.56	0.67	0.71	Address (64)	0.21	0.54	0.70	0.77
PC (4)	0.29	0.61	0.73	0.78	PC (32)	0.18	0.52	0.72	0.82
AddrAndPC (4)	0.34	0.55	0.61	0.63	AddrAndPC (32)	0.31	0.62	0.74	0.78
AddrOrPC (4)	0.28	0.60	0.74	0.80	AddrOrPC (32)	0.16	0.48	0.70	0.80
Hybrid (4)	0.34	0.64	0.74	0.78	Hybrid (32)	0.28	0.64	0.79	0.85

Table 6.1: F_β values for predictors.

two tables each, are also configured with each table of size 4. It can be seen that the Address and AddressAndPC predictors do comparatively well according to the F_1 measure, since they have high Precision. When Sensitivity is prioritised, these two predictors do noticeably worse than the others. The hybrid bimodal table has the best performance on almost all the measures.

This comparative performance leads to the conclusion that if area and power are at a premium, then a simple Program Counter based buffer is the best predictor to use. It is clear that the PC based predictor does better in both Sensitivity and Precision than the address based one. Combined with easier implementation (as described in Section 4.4), this makes the PC predictor the clear choice over the address predictor. In Figure 6.7j it can be seen that the address based predictor, provides the worst tradeoff among the predictors evaluated, giving the poorest Precision for any given Sensitivity.

Otherwise, if area and power are not tightly constrained, a hybrid bimodal predictor combining Program Counter and address buffers is the best choice.

6.2 Checkpointing Scheme

It would be desirable to find a limit for the maximum savings available through checkpointing, however, it is not straightforward to do so. Firstly, placing checkpoints through an oracle predictor would not necessarily lead to the best possible placement of checkpoints. Further, since placing a checkpoint effects execution, and hence may change the relative ordering of loads of stores and the resulting dependence violations, it is not possible to use a trace of execution as a perfect predictor, as is done when evaluating branch predictors. Constructing a checkpointing scheme that uses an oracle predictor would involve the following: whenever a dependence is observed, rewinding execution at least as far back as the load involved in the dependence, and placing a checkpoint at that load. In the infrastructure used for evaluation, this means either restarting execution on every dependence or saving the entire state of the simulator periodically. For the purpose of this evaluation, it was felt that such a methodology would be too complex and time consuming, specially since it would not provide a precise limit on improvement through checkpointing, and a heuristic would still need to

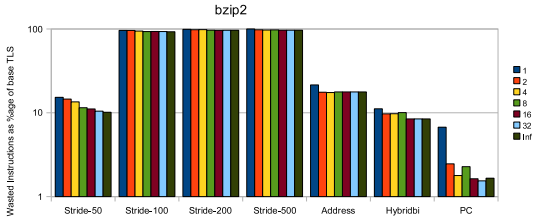
be employed to avoid checkpoints from being placed excessively close.

In this section some of the predictors described earlier are used to place checkpoints, as well as placing checkpoints by stride, and the results are evaluated. Checkpointing policies are evaluated by observing the effect of checkpointing on the number of wasted instructions. The wasted re-execution is shown as a percentage of the wasted re-execution in the case without any checkpointing. For the address and PC predictors, tables of size 32 are used, and the hybrid bimodal employs a 128 entry, 5 bit metatable.

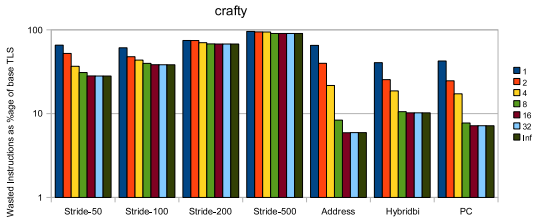
For the first set of results, in Figure 6.8 a checkpoint is placed whenever a positive dependence prediction is obtained. The wasted instructions are shown for different checkpointing schemes and for different values for the maximum number of checkpoints allowed per task. When the maximum number of checkpoints is reached, all further positive predictions are ignored. It should be noted that the vertical axis in Figure 6.8 is in *log* scale. To keep the figure easy to read, the vertical axis starts at 1, so in cases where the wasted instructions are less than 1% of non-checkpointed TLS, no bar appears.

The savings from checkpointing by stride are much less than using dependence predictors. Among the dependence predictors, the PC predictor produces the most savings, followed by the hybrid, and then the address. Only in one benchmark, *vortex*, does checkpointing by address result in more savings than doing so by PC. This is consistent with the predictor Sensitivity shown in Figure 6.7, where the PC predictor has higher sensitivity for 32 entries for all benchmarks apart from *vortex*. The address and hybrid predictors show improved savings up to 16 checkpoints, while the PC predictor shows improvement up to 32. This may be a reflection of the lower Precision of the PC predictor, which leads to a need for more checkpoints to achieve the maximum saving.

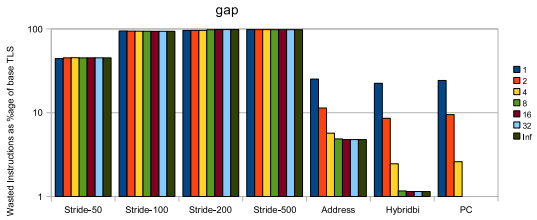
In Figure 6.9a, the reduction in wasted instructions for address, Program Counter, hybrid and stride based checkpointing is summarised. For this, a maximum of 8 checkpoints per task are assumed. Predictor based checkpoints behave consistently with the results in the previous section. The PC predictor provides the most reduction, followed by the hybrid bimodal and address predictor. This is in keeping with the Sensitivity observed for these predictors. In Figure 6.9b, the efficiency of checkpoints is shown in terms of the number of instructions saved per checkpoint plotted against the wasted


bzip2
Wasted Instructions as %age of base TLS
100
10
1
Stride-50
Stride-100
Stride-200
Stride-500
Address
Hybridbi
PC
1
2
4
8
16
32
Inf

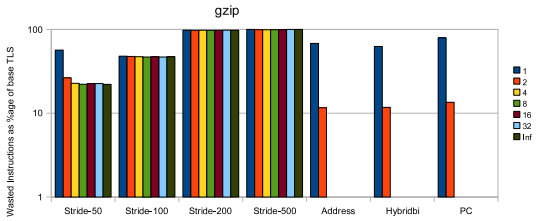
(a)


crafty
Wasted Instructions as %age of base TLS
100
10
1
Stride-50
Stride-100
Stride-200
Stride-500
Address
Hybridbi
PC
1
2
4
8
16
32
Inf

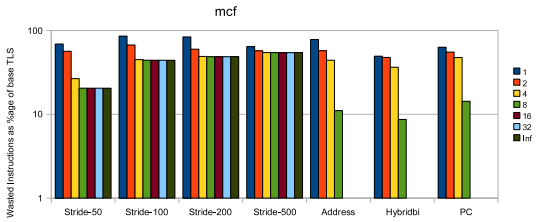
(b)


gap
Wasted Instructions as %age of base TLS
100
10
1
Stride-50
Stride-100
Stride-200
Stride-500
Address
Hybridbi
PC
1
2
4
8
16
32
Inf

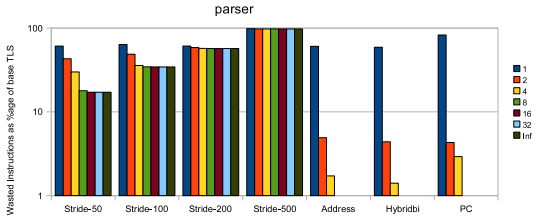
(c)


gzip
Wasted Instructions as %age of base TLS
100
10
1
Stride-50
Stride-100
Stride-200
Stride-500
Address
Hybridbi
PC
1
2
4
8
16
32
Inf

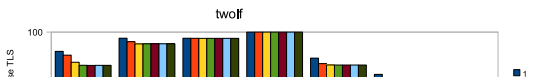
(d)

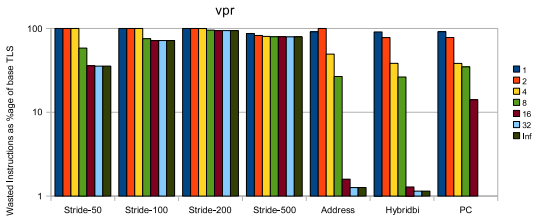

mcf
Wasted Instructions as %age of base TLS
100
10
1
Stride-50
Stride-100
Stride-200
Stride-500
Address
Hybridbi
PC
1
2
4
8
16
32
Inf

(e)

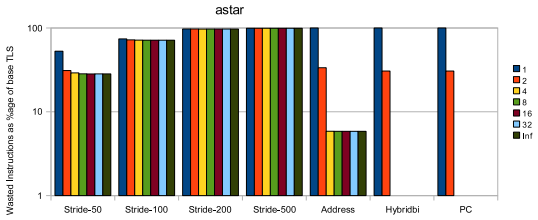

parser
Wasted Instructions as %age of base TLS
100
10
1
Stride-50
Stride-100
Stride-200
Stride-500
Address
Hybridbi
PC
1
2
4
8
16
32
Inf

(f)

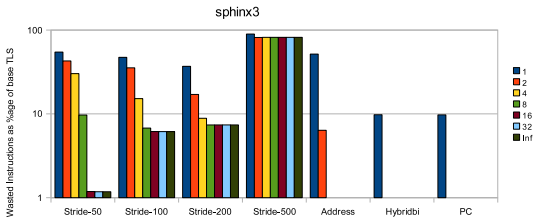

twolf
100



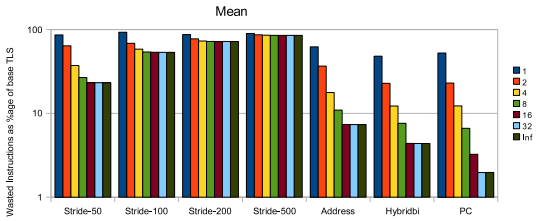
(i)



(j)

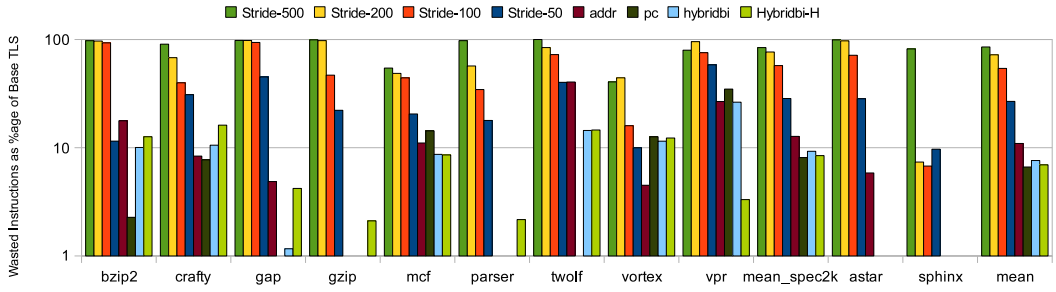


(k)

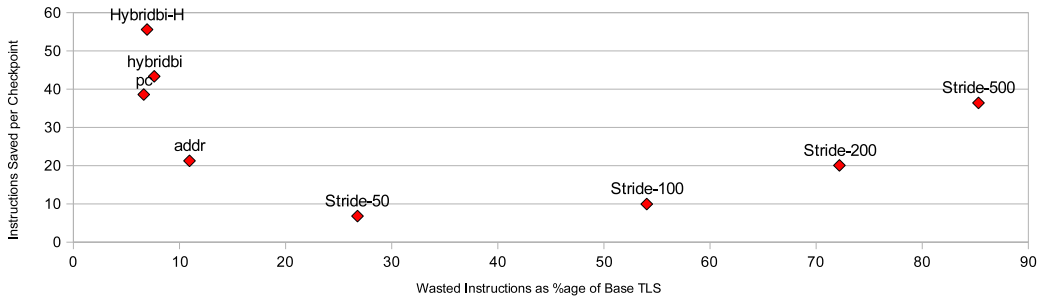


(l)

Figure 6.8: Continued: Change in savings of wasted instructions as the maximum number of checkpoints is changed.



(a) Wasted instructions for checkpointing using different placement schemes.



(b) Wasted instructions vs. savings per checkpoint.

Figure 6.9: Unnecessary re-execution as a percentage of TLS with no checkpointing for different checkpoint placement schemes.

re-execution. Here, it can be seen that the address based predictor, apart from having the lowest savings among the three predictor types, is also the least efficient in terms of savings per checkpoint, saving only an average of 20 instructions per checkpoint. The PC based predictor gets better savings with greater efficiency than the address based predictor. The hybrid bimodal predictor places checkpoints with the highest efficiency, and obtains a reduction that lies between that for the PC and address predictors. Hybridbi-H shows the case where the heuristic described in Section 4.3 is used rather than inserting a checkpoint on every positive prediction. It can be seen that using this policy that takes into account the limitation on the number of checkpoints provides improved efficiency in terms of savings per checkpoint. The saved re-execution per-checkpoint increases by 30% when using this policy over checkpointing on each positive prediction. There is also a small reduction in wasted re-execution,

from 8.4% to 7.6%.

The predictors are also compared against stride checkpointing. The performance for checkpoints inserted by stride is far lower than that for using predictors. Small strides result in reducing wasted re-execution, but with very small savings per checkpoint. Larger strides do not produce much saving.

The saving per checkpoint for all the schemes is fairly low - the highest, for the hybrid bimodal predictor, is an average of 56 instructions per checkpoint. For some programs, such as *crafty* and *mcf*, this value is less than 20. This means, that to obtain any advantage from checkpointing for these programs, it is important that the checkpointing mechanism has low overhead.

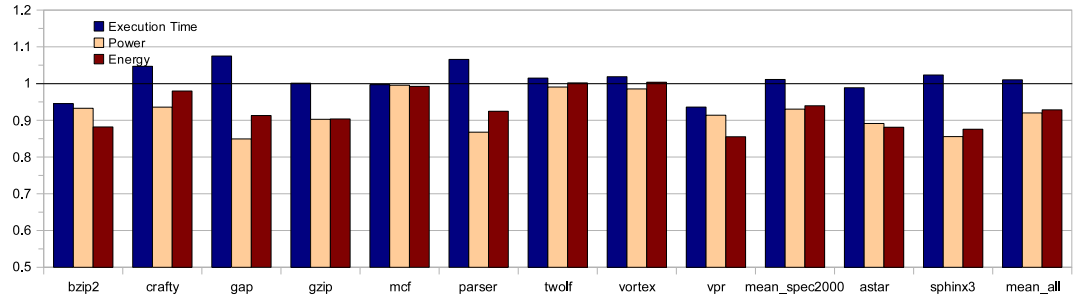


Figure 6.10: Checkpointing shows power improvement, resulting in an energy improvement of 7% on average over base TLS.

The effects on execution time and power of checkpointing are shown in Figure 6.10. The figure shows execution time, power and energy for checkpointed execution normalised against base TLS. For this evaluation we use a hybrid bimodal dependence predictor for inserting checkpoints, and selective restart is employed. The tasks for the SPEC 2000 benchmarks have been selected through a profiler. This has resulted in tasks with late dependences being pruned out. This results in a low ratio of wasted instructions to committed instructions. Even though few tasks in these benchmarks are good candidates for checkpointing, there is still a 6% improvement in energy on average, with up to 14% for *vpr*, 10% for *gzip* and 9% for *gap*. The source of this is an improvement in power, with a negligible effect on execution time on average. We do observe some speedups and slowdowns for individual programs. The worst slowdowns

are for *gap* and *parser*, while *bzip* and *vpr* see substantial speedups.

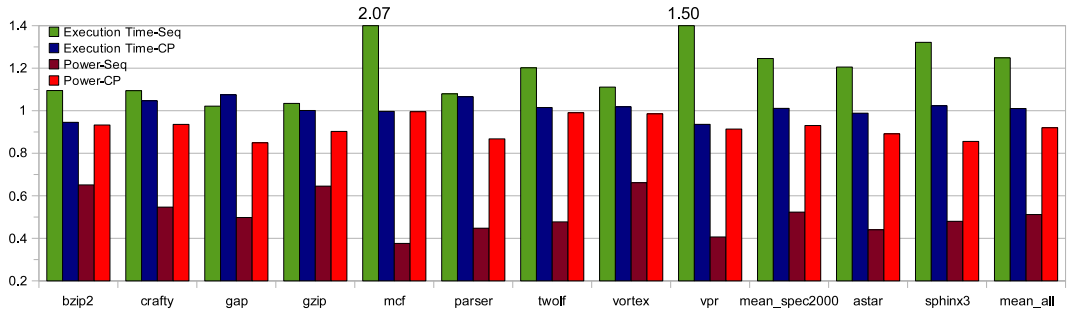


Figure 6.11: Checkpointing recovers some of the power lost to speculation while maintaining speedup.

Figure 6.11 puts these results in the context of sequential execution. Once again, execution time and power are normalised against base TLS execution. It is clear that the performance improvement of TLS is obtained at the cost of greatly increased power. Even in cases such as *gap* and *gzip*, where the speedup through speculation is very small, the power increases substantially. Checkpointing is able to recover some of this lost power with a negligible effect on execution time.

astar and *sphinx3* from SPEC2006 have much longer running tasks than any of the SPEC2000 benchmarks. In particular, *sphinx* regularly shows violations due to loads over 450 instructions from a task boundary. This presents a good opportunity for energy savings through checkpointing. *astar* has greater variation in task sizes and how far into tasks violating loads occur. This makes it a good candidate for checkpointing as well. For all the benchmarks evaluated, checkpointing results in a 7% energy improvement on average.

6.3 Sensitivity to Architectural Extentions

6.3.1 Memory System Modification

Checkpointing results in added pressure on the versioned memory system. This has been discussed in Section 3.2.2. In the applications considered, because the working

sets are small, the extra restarts due to failed allocation of speculative space have a very small effect on execution time and power. The reason is that the working sets are small, and even for checkpointed execution, the number of restarts due to failed allocation is very small compared to the total number of restarts (less than 3% of total restarts for all benchmarks and less than 1% on average). For programs with large working sets, and particularly those with long running speculative tasks, this could become a performance concern.

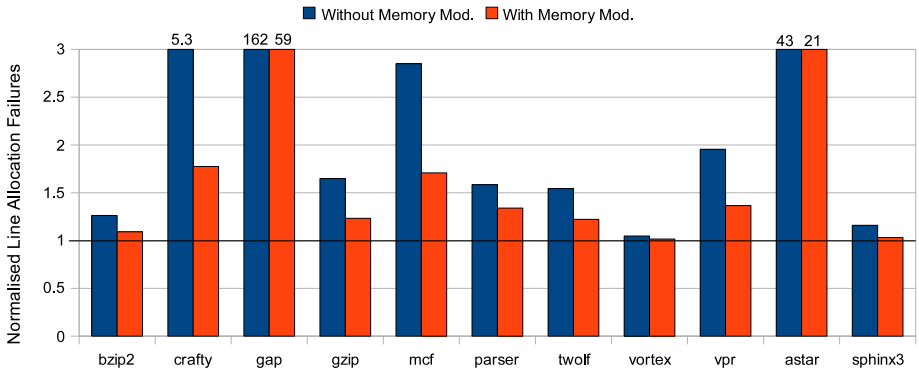


Figure 6.12: Number of times allocation of a speculative line fails in checkpointed execution normalised against TLS without checkpointing.

To measure the effect of the changes described in Section 3.2.2, the number of failed allocations is shown in Figure 6.12. As before, checkpoints are inserted using a hybrid bimodal dependence predictor. It can be seen that for many benchmarks there is a large increase in the number of failed allocations when checkpointing is employed. The changes proposed reduce the number of additional failed allocations due to restarts to less than a third on average.

6.3.2 Restart Mechanism

In Figure 6.13, we show the effect of selective restart, as described in Section 3.2.1, on execution time. We note that the programs showing substantial improvement in execution time due to checkpointing (*bzip*, *vpr* and *astar*) only do so when selective restart is used. The selective restart mechanism also reduces the execution time penalty on *gap*, *parser* and *sphinx3*. The benchmarks that see little effect on execution time

from checkpointing consequently see little effect from selective restart. The mean improvement in execution time due to selective restart is 3%, with up to 8% for *bzip2* and *astar*, and 5% for *vpr*.

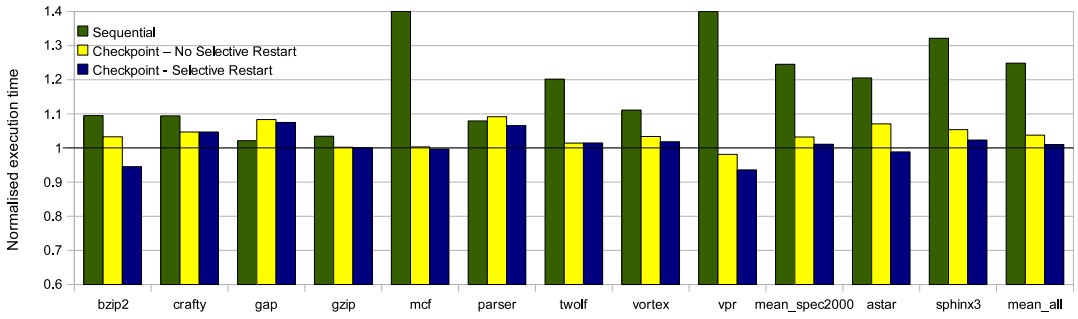
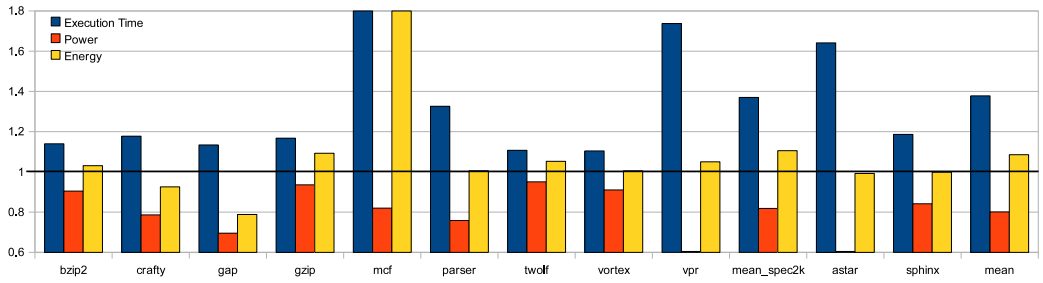


Figure 6.13: Execution time for sequential execution and TLS when checkpointing with and without selective restart, normalised against TLS without checkpoints.

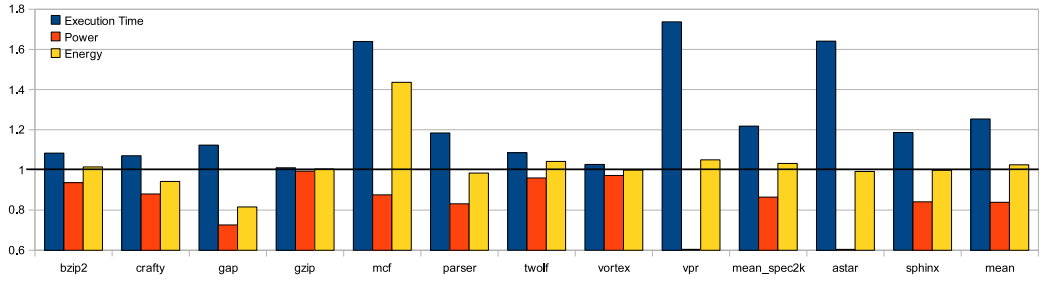
6.4 Using Dependence Prediction for Synchronisation

An alternative to checkpointing for avoiding wasteful re-execution is synchronisation. In this section, a brief demonstration is made of the application of the dependence prediction techniques described earlier towards synchronisation. However, this section does not intend to perform a detailed evaluation of which prediction technique is best for synchronisation.

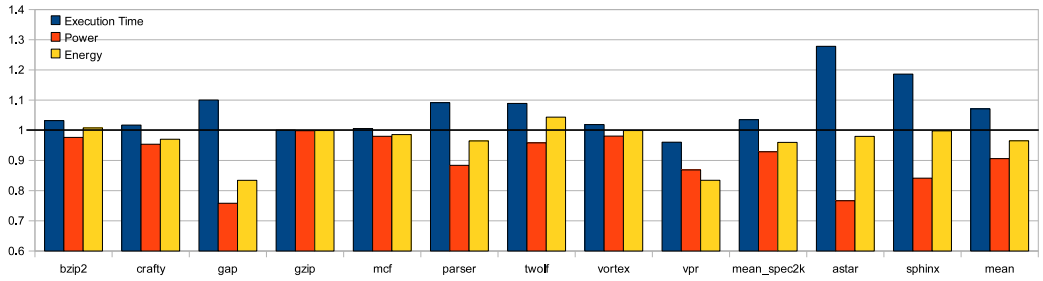
Figure 6.14 shows the results observed when different policies are applied to synchronisation. First, in Figure 6.14a, a predictor similar to the one used for checkpointing is used. This is a 1024 entry, 5 bit hybrid bimodal predictor, with 32 entry Address and PC Criticality Tables and a 64 entry PC Translation table. It can be seen that the performance of the system is degraded significantly when synchronising. This is because the predictor has a low Precision. Every false positive causes unnecessary synchronisation and results in most computation being serialised. This is reflected in a power improvement of 20% but a slowdown of 38% on average, which results in a 9% increase in energy consumed. The slowdown is large enough to make the system



(a) Aggressively synchronising dependences.



(b) Synchronising dependences with a less aggressive predictor.



(c) More conservative synchronisation, adding minimum task size.

Figure 6.14: Effect of synchronisation on execution time, power and energy, normalised against base TLS execution.

slower than non-TLS sequential execution.

In Figure 6.14b, the predictor is configured to be less aggressive, with higher Precision. In this case, the bits per entry in the metatable are reduced to 2, and the Address and PC Criticality tables are reduced to 2 entries each. It can be seen that this greatly improves the performance of the system when synchronising. The execution

time degradation is reduced to 25% and the increase in energy to 3%. The fact that there is a significant slowdown and an energy degradation suggests that the system is still synchronising too often.

Figure 6.14c shows the results when the synchronisation is made even more conservative, by changing the policy so that synchronisation on a dependence prediction only takes place if the size of the task is at least 50 instruction. This results in synchronisation now showing an Energy improvement of 3%, but still a slowdown of 7%.

These results show that it is crucial to have a well performing prediction mechanism for synchronisation to be effective. Further, since there is considerable cost to unnecessary synchronisation in the form of serialisation, the Precision of the predictor is far more important.

Chapter 7

Related Work

7.1 Thread Level Speculation

Thread level speculation has been previously proposed (e.g., (14; 16; 21; 38; 40)) as a means to provide some degree of parallelism in the presence of data dependencies. The vast majority of prior work on TLS systems has focused on architectural features directly related to the TLS support, such as protocols for multi-versioned caches and data dependence violation detection. All these are orthogonal to our work. In particular, we use the system proposed by Renau et al. (34) as our baseline.

7.2 Checkpointing

This thesis uses checkpoints to tolerate dependences between speculative tasks. The most directly related previous work is by Colohan et al. (8) and Waliullah and Stenstrom (49). The work in (8) proposes checkpointing as part of extensions to the TLS mechanism to support long running threads. Checkpoints are placed at fixed instruction strides without any prediction of the dependence behaviour of instructions or memory addresses. We find that this approach is not suitable for our applications and does not produce significant savings. We apply checkpoints to smaller tasks as well, which means the overhead of checkpoints is not negligible and so smarter placement schemes are required. The work by Waliullah and Stenstrom (49) looks at intermediate check-

points to improve behaviour for transactions in a Transactional Memory system.

Checkpointing is also used to aid finer grained speculation in processors. They are used to recover from mispredicted branches. CAVA (3) uses checkpoints to assist value prediction on *L2* cache misses. On an *L2* miss, a checkpoint is placed and execution continues with a predicted value. CHERRY (23) uses checkpoints to allow early recycling of resources, by decoupling resource release and instruction retirement.

There is extensive work in the area of using checkpoints as part of fault tolerance schemes. A survey of the area is provided by Elnozahy et al. (10). In particular, Wu et al. (51) use a mechanism similar to that used for TLS, by tagging cache blocks with checkpoint IDs. Sorin et al. (39) use checkpoint schemes to support long latency fault detection schemes in shared memory multiprocessors.

7.3 Other Schemes for Reducing Wasted Execution

Other proposals have been made to tolerate dependences between tasks through learning dependences and dynamically synchronising to avoid violations (7; 42). Zhai et al. (55) statically synchronise scalar communication at compile time. These have been discussed in some detail in Section 2.2.1.

A different mechanism for selective re-execution is to find the *slice* of instructions affected by a dependence violation (35). Tuck and Tullsen (46) use multiple contexts to recover from failed value prediction.

7.4 Data Dependence Prediction

Data dependence prediction has been previously proposed in various contexts. Moshovos and Sohi (26) use dependence prediction to identify loads and stores that are dependent via memory operations. These predictions are used to speculatively execute dependent loads without waiting for memory operations to complete. Predictions are also used to leverage a small *Transient Value Cache*, avoiding accessing the data cache for short lived values. Similarly, Chrysos et al.(5) predict dependences through tracking a *store set* for each load in order to speculatively execute the load as soon as possible.

Waliullah and Stenstrom (49), in the use of dependence prediction most directly related to this thesis, use a technique very similar to the address based scheme discussed. These predictions are used to checkpoint transactions in a Transactional Memory system. We observe that in many cases, address based predictors fail to find dependent loads.

Cintra and Torrellas (7) and Steffan et al. (42) use dependence prediction to synchronise speculative tasks. Xekalakis et al. (53) use dependence predictions to estimate the likelihood of squashes for speculative tasks in order to allocate resources.

In other work (53), we have used data dependence prediction to estimate whether a task is performing useful work and scale the voltage and frequency of the processor accordingly.

Chapter 8

Conclusions and Future Work

This chapter presents the conclusions reached and discusses possibilities for extension of the work presented in this thesis.

8.1 Summary of Contributions

This thesis makes two sets of contributions: mechanisms for efficient checkpointing and dependence prediction techniques.

The thesis improves the efficiency of TLS systems by crafting efficient checkpointing. This is done through extending the base TLS protocol with selective restart and making changes to the versioned memory system. It is shown that selective restart is important for maintaining the execution time advantage of TLS when checkpointing. A heuristic for placing checkpoints based on depedence prediction is proposed.

An evaluation of various dependence prediction techniques is performed and showed that Program Counter based and hybrid predictors outperform earlier proposals. To our knowledge there has been no previous comparison of dependence prediction techniques for coarse grained speculation. The evaluation also shows that using dependence prediction is a far more efficient way of placing checkpoints than doing so by stride as proposed previously. The practical issues associated with constructing predictors are also discussed, with the conclusion that Program Counter based predictors present fewer complications in implementation than previously proposed address based

predictors.

It is concluded that checkpointing based on dependence prediction is an effective way of reducing inefficiency in speculative execution. Using the checkpoint mechanisms proposed, and placement policy based on dependence prediction, the benchmarks evaluated show energy improvement of up to 14% , and 7% on average. This is achieved with a very small effect (1%) on execution time. Comparing checkpointing with synchronisation shows that synchronisation achieves energy improvements but at a substantial cost in execution time.

The presence of intermediate checkpoints makes misspeculation far less expensive. This changes the tradeoffs in task selection. We believe that it makes the problem of task selection easier since more aggressive speculation can be performed, and profile based task pruning is less critical to performance than in previous proposals.

8.2 Future Work

There are many avenues of future extension based on this thesis. These include improvements in dependence prediction, checkpointing policy, and integrating with wider systems.

The dependence prediction techniques evaluated in this thesis do not use any information beyond the violation history for addresses and instructions. The predictors achieve good Sensitivity but fairly low Precision. Using more information (context, stride) for prediction may be able to improve predictor Precision.

More sophisticated checkpointing policies may be able to improve performance, by taking into account detailed resource information, such as the pressure on the speculative cache, or tracking task sizes to more intelligently adapt the size of tasks before they are checkpointing.

Checkpointing can be combined with value prediction. Predicting speculatively used values can reduce the probability of dependence violations, but can still have a high cost in case of misprediction. This cost can be reduced by checkpointing when a value is predicted.

Checkpointing and synchronisation can also be employed at the same time. For

instance, if the dependence predictor can provide a confidence measure, then synchronisation is a better choice in the case where a dependence is predicted with high confidence. However, if a dependence is predicted with low confidence, then a checkpoint can be placed. A more sophisticated system can take into account effects on the memory system, so that even if a dependence is predicted with high confidence, the system may continue checkpointed execution in order to obtain benefit from prefetching.

Checkpointing has repercussions on task selection. An area of future exploration is establishing what changes need to be made to the task selection algorithm to get optimum preformance.

Bibliography

[1] BHOWMIK, A., AND FRANKLIN, M. A general compiler framework for speculative multithreading. In *SPAA '02: Proceedings of the fourteenth annual ACM Symposium on Parallel Algorithms and Architectures* (New York, NY, USA, 2002), ACM Press, pp. 99–108.

[2] BROOKS, D., TIWARI, V., AND MARTONOSI, M. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA '00: Proceedings of the 27th annual International Symposium on Computer Architecture* (2000).

[3] CEZE, L., STRAUSS, K., TUCK, J., TORRELLAS, J., AND RENAU, J. Cava: Using checkpoint-assisted value prediction to hide L2 misses. *ACM Transactions on Architecture and Code Optimization 3*, 2 (2006), 182–208.

[4] CHEN, P.-S., HUNG, M.-Y., HWANG, Y.-S., JU, R. D.-C., AND LEE, J. K. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2003), ACM, pp. 25–36.

[5] CHRYSOS, G. Z., AND EMER, J. S. Memory dependence prediction using store sets. In *ISCA '98: Proceedings of the 25th annual International Symposium on Computer architecture* (Washington, DC, USA, 1998), IEEE Computer Society, pp. 142–153.

[6] CINTRA, M., AND LLANOS, D. R. Toward efficient and robust software speculative parallelization on multiprocessors. In *PPoPP '03: Proceedings of the ninth*

ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (New York, NY, USA, 2003), ACM, pp. 13–24.

[7] CINTRA, M., AND TORRELLAS, J. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture* (Feb. 2002), pp. 43–54.

[8] COLOHAN, C. B., AILAMAKI, A., STEFFAN, J. G., AND MOWRY, T. C. Tolerating dependences between large speculative threads via sub-threads. In *ISCA '06: Proceedings of the 33rd annual International Symposium on Computer Architecture* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 216–226.

[9] DOU, J., AND CINTRA, M. Compiler estimation of load imbalance overhead in speculative parallelization. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 203–214.

[10] ELNOZAHY, E. N. M., ALVISI, L., WANG, Y.-M., AND JOHNSON, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys 34*, 3 (2002), 375–408.

[11] EVERS, M., CHANG, P.-Y., AND PATT, Y. N. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. In *ISCA '96: Proceedings of the 23rd annual International Symposium on Computer architecture* (New York, NY, USA, 1996), ACM, pp. 3–11.

[12] FRANKLIN, M., AND SOHI, G. S. Arb: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers 45*, 5 (1996), 552–571.

[13] GARZARÁN, M. J., PRVULOVIC, M., LLABERÍA, J. M., VIÑALS, V., RAUCHWERGER, L., AND TORRELLAS, J. Tradeoffs in buffering memory state for thread-level speculation in multiprocessors. *HPCA '03: Proceedings of the 9th*

International Symposium on High-Performance Computer Architecture (2003), 191.

[14] HAMMOND, L., WILLEY, M., AND OLUKOTUN, K. Data speculation support for a chip multiprocessor. In *ASPLOS-VIII: Proceedings of the eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 1998), ACM, pp. 58–69.

[15] JOHNSON, T. A., EIGENMANN, R., AND VIJAYKUMAR, T. N. Min-cut program decomposition for thread-level speculation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation* (New York, NY, USA, 2004), ACM Press, pp. 59–70.

[16] KRISHNAN, V., AND TORRELLAS, J. Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In *ICS '98: Proceedings of the 12th International Conference on Supercomputing* (New York, NY, USA, 1998), ACM, pp. 85–92.

[17] LAM, M. S., AND WILSON, R. P. Limits of control flow on parallelism. In *ISCA '92: Proceedings of the 19th annual International Symposium on Computer Architecture* (New York, NY, USA, 1992), ACM Press, pp. 46–57.

[18] LIU, W., TUCK, J., CEZE, L., AHN, W., STRAUSS, K., RENAU, J., AND TORRELLAS, J. Posh: a TLS compiler that exploits program structure. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2006), ACM, pp. 158–167.

[19] LUO, Y., PACKIRISAMY, V., HSU, W.-C., ZHAI, A., MUNGRE, N., AND TARKAS, A. Dynamic performance tuning for speculative threads. In *ISCA '09: Proceedings of the 36th annual International Symposium on Computer architecture* (New York, NY, USA, 2009), ACM, pp. 462–473.

[20] MADRILES, C., GARCÍA-QUIÑONES, C., SÁNCHEZ, J., MARCUELLO, P., GONZÁLEZ, A., TULLSEN, D. M., WANG, H., AND SHEN, J. P. Mitosis:

A speculative multithreaded processor based on precomputation slices. *IEEE Transactions on Parallel and Distributed Systems 19*, 7 (2008), 914–925.

[21] MARCUELLO, P., AND GONZÁLEZ, A. Clustered speculative multithreaded processors. In *ICS '99: Proceedings of the 13th International Conference on Supercomputing* (New York, NY, USA, 1999), ACM, pp. 365–372.

[22] MARCUELLO, P., TUBELLA, J., AND GONZÁLEZ, A. Value prediction for speculative multithreaded architectures. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE International Symposium on Microarchitecture* (Washington, DC, USA, 1999), IEEE Computer Society, pp. 230–236.

[23] MARTÍNEZ, J. F., RENAU, J., HUANG, M. C., PRVULOVIC, M., AND TORRELLAS, J. Cherry: checkpointed early resource recycling in out-of-order microprocessors. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE International Symposium on Microarchitecture* (Los Alamitos, CA, USA, 2002), IEEE Computer Society Press, pp. 3–14.

[24] MCFARLING, S. Combining branch predictors. Tech. Rep. WRL Technical Note TN-36, Western Research Laboratory, June 1993.

[25] MOSHOVOS, A., BREACH, S. E., VIJAYKUMAR, T. N., AND SOHI, G. S. Dynamic speculation and synchronization of data dependences. In *ISCA '97: Proceedings of the 24th annual International Symposium on Computer Architecture* (New York, NY, USA, 1997), ACM, pp. 181–193.

[26] MOSHOVOS, A., AND SOHI, G. S. Streamlining inter-operation memory communication via data dependence prediction. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE International Symposium on Microarchitecture* (Washington, DC, USA, 1997), IEEE Computer Society, pp. 235–245.

[27] OLUKOTUN, K., NAYFEH, B. A., HAMMOND, L., WILSON, K., AND CHANG, K. The case for a single-chip multiprocessor. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural Support for Programming*

Languages and Operating Systems (New York, NY, USA, 1996), ACM Press, pp. 2–11.

[28] OPLINGER, J. T., HEINE, D. L., AND LAM, M. S. In search of speculative thread-level parallelism. In *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques* (Washington, DC, USA, 1999), IEEE Computer Society, p. 303.

[29] PACKIRISAMY, V., ZHAI, A., HSU, W.-C., YEW, P.-C., AND NGAI, T.-F. Exploring speculative parallelism in spec2006. In *ISPASS '09: Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software* (Boston, Massachusetts, USA, April 2009), IEEE Computer Society, pp. 77–88.

[30] QUINONES, C. G., MADRILES, C., SÁNCHEZ, J., MARCUELLO, P., GONZÁLEZ, A., AND TULLSEN, D. M. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation* (New York, NY, USA, 2005), ACM, pp. 269–279.

[31] RAMAN, E., VACHHARAJANI, N., RANGAN, R., AND AUGUST, D. I. Spice: speculative parallel iteration chunk execution. In *CGO '08: Proceedings of the international symposium on Code Generation and Optimization* (New York, NY, USA, 2008), ACM, pp. 175–184.

[32] RENAU, J. *Chip multiprocessors with speculative multithreading: design for performance and energy efficiency*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 2004. Adviser-Josep Torrellas.

[33] RENAU, J., FRAGUELA, B., TUCK, J., LIU, W., PRVULOVIC, M., CEZE, L., SARANGI, S., SACK, P., STRAUSS, K., AND MONTESINOS, P. SESC simulator, January 2005. http://sesc.sourceforge.net.

[34] RENAU, J., TUCK, J., LIU, W., CEZE, L., STRAUSS, K., AND TORRELLAS, J. Tasking with out-of-order spawn in tls chip multiprocessors: microarchitecture

and compilation. In *ICS '05: Proceedings of the 19th International Conference on Supercomputing* (New York, NY, USA, 2005), ACM Press, pp. 179–188.

[35] SARANGI, S. R., WEI LIU, J. T., AND ZHOU, Y. Reslice: Selective re-execution of long-retired misspeculated instructions using forward slicing. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 257–270.

[36] SEZNEC, A. Analysis of the o-geometric history length branch predictor. In *ISCA '05: Proceedings of the 32nd annual International Symposium on Computer Architecture* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 394–405.

[37] SILVA, J. D., AND STEFFAN, J. G. A probabilistic pointer analysis for speculative optimizations. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2006), ACM, pp. 416–425.

[38] SOHI, G. S., BREACH, S. E., AND VIJAYKUMAR, T. N. Multiscalar processors. In *ISCA '95: Proceedings of the 22nd annual International Symposium on Computer Architecture* (New York, NY, USA, 1995), ACM, pp. 414–425.

[39] SORIN, D. J., MARTIN, M. M. K., HILL, M. D., AND WOOD, D. A. Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *ISCA '02: Proceedings of the 29th annual International Symposium on Computer Architecture* (Washington, DC, USA, 2002), IEEE Computer Society, pp. 123–134.

[40] STEFFAN, J., AND MOWRY, T. The potential for using thread-level data speculation to facilitate automatic parallelization. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture* (Washington, DC, USA, 1998), IEEE Computer Society, p. 2.

[41] STEFFAN, J. G., COLOHAN, C. B., ZHAI, A., AND MOWRY, T. C. A scalable approach to thread-level speculation. In *ISCA '00: Proceedings of the 27th an-*

nual International Symposium on Computer Architecture (New York, NY, USA, 2000), ACM, pp. 1–12.

[42] STEFFAN, J. G., COLOHAN, C. B., ZHAI, A., AND MOWRY, T. C. Improving value communication for thread-level speculation. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture* (Washington, DC, USA, 2002), IEEE Computer Society, p. 65.

[43] TARJAN, D., THOZIYOOR, S., AND JOUPPI, N. P. Cacti 4.0. Tech. rep., Compaq Western Research Lab., 2006.

[44] TSAI, J.-Y., JIANG, Z., AND YEW, P.-C. Compiler techniques for the superthreaded architectures. *International Journal of Parallel Programming 27*, 1 (1999), 1–19.

[45] TSAI, J.-Y., AND YEW, P.-C. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *PACT '96: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques* (Washington, DC, USA, 1996), IEEE Computer Society, pp. 35–47.

[46] TUCK, N., AND TULLSEN, D. M. Multithreaded value prediction. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 5–15.

[47] VEENSTRA, J. E., AND FOWLER, R. J. Mint: A front end for efficient simulation of shared-memory multiprocessors. In *MASCOTS '94: Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems* (Washington, DC, USA, 1994), IEEE Computer Society, pp. 201–207.

[48] VIJAYKUMAR, T. N., AND SOHI, G. S. Task selection for a multiscalar processor. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture* (Los Alamitos, CA, USA, 1998), IEEE Computer Society Press, pp. 81–92.

[49] WALIULLAH, M., AND STENSTROM, P. Intermediate checkpointing with conflicting access prediction in transactional memory systems. In *IPDPS '08: Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing* (April 2008), pp. 1–11.

[50] WANG, Y., AN, H., LIANG, B., WANG, L., CONG, M., AND REN, Y. Balancing thread partition for efficiently exploiting speculative thread-level parallelism. In *APPT '07: Sixth International Workshop on Advanced Parallel Processing Technologies* (2007), pp. 40–49.

[51] WU, K. L., FUCHS, W. K., AND PATEL, J. H. Error recovery in shared memory multiprocessors using private caches. *IEEE Transactions on Parallel and Distributed Systems 1*, 2 (1990), 231–240.

[52] XEKALAKIS, P., IOANNOU, N., AND CINTRA, M. Combining thread level speculation helper threads and runahead execution. In *ICS '09: Proceedings of the 23rd International Conference on Supercomputing* (New York, NY, USA, 2009), ACM, pp. 410–420.

[53] XEKALAKIS, P., IOANNOU, N., KHAN, S., AND CINTRA, M. Profitability-based power allocation for speculative multithreaded systems. In *IPDPS '10: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium* (2010).

[54] ZHAI, A. *Compiler optimization of value communication for thread-level speculation*. PhD thesis, Pittsburgh, PA, USA, 2005.

[55] ZHAI, A., COLOHAN, C. B., STEFFAN, J. G., AND MOWRY, T. C. Compiler optimization of scalar value communication between speculative threads. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2002), ACM, pp. 171–183.

[56] ZHAI, A., COLOHAN, C. B., STEFFAN, J. G., AND MOWRY, T. C. Compiler optimization of memory-resident value communication between specula-

tive threads. In *CGO '04: Proceedings of the international symposium on Code Generation and Optimization* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 39–50.