Abstract

ANANTARAMAN, ARAVINDH VENKATASESHADRI
Reducing Frequency in Real-Time Systems via Speculation and Fall-Back Recovery
(Under the direction of Dr. Eric Rotenberg)

In real-time systems, safe operation requires that tasks complete before their deadlines. Static worst-case timing analysis is used to derive an upper bound on the number of cycles for a task, and this is the basis for a safe frequency that ensures timely completion in any scenario. Unfortunately, it is difficult to tightly bound the number of cycles for a complex task executing on a complex pipeline, and so the safe frequency tends to be over-inflated. Power efficiency is sacrificed for safety.

The situation only worsens as advanced microarchitectural techniques are deployed in embedded systems. High-performance microarchitectural techniques such as caching, branch prediction, and pipelining decrease typical execution times. At the same time, it is difficult to tightly bound the worst-case execution time of complex tasks on highly dynamic substrates. As a result, the gap between worst-case execution time and typical execution time is expected to increase.

This thesis explores frequency speculation, a technique for reconciling the power/safety trade-off. Tight but unsafe bounds (derived from past task executions) are the basis for a low speculative frequency. The task is divided into multiple smaller sub-tasks and each sub-task is assigned an interim soft deadline, called a checkpoint. Sub-tasks are attempted at the speculative frequency. Continued safe progress of the task as a whole is confirmed for as long as speculative sub-tasks complete before their checkpoints. If a sub-task exceeds its checkpoint (misprediction), the system falls back

to a higher recovery frequency that ensures the overall deadline is met in spite of the interim misprediction.

The primary contribution of this thesis is the development of two new frequency speculation algorithms. A drawback of the original frequency speculation algorithm is that a sub-task misprediction is detected only after completing the sub-task. The misprediction can be detected earlier through the use of a watchdog timer that expires at the checkpoint unless the sub-task completes in time to advance it to the next checkpoint. Early detection is superior because recovery can be initiated earlier, in the middle of the mispredicted sub-task. This introduces extra slack that can be used to lower the speculative frequency even further.

A new issue that arises with early detection is bounding the amount of work that remains in the mispredicted sub-task after the misprediction is detected. The two new algorithms differ in how the unfinished work is bounded. The first algorithm conservatively bounds the execution time of the unfinished portion using the worst-case execution time of the entire sub-task. The second uses more sophisticated analysis to derive a much tighter bound. Both early-detection algorithms outperform the late-detection algorithm. For tight deadlines, the sophisticated analysis of the second early-detection algorithm truly pays off. It yields 60-70% power savings for six real-time applications from the C-lab suite.

Reducing Frequency in Real-Time Systems via Speculation and Fall-Back Recovery

by

Aravindh Venkataseshadri Anantaraman

A thesis submitted to the graduate faculty of

North Carolina State University

In partial fulfillment of the requirements of the degree of

Master of Science

COMPUTER ENGINEERING

Raleigh

2003

Approved by

_____

Dr. Eric Rotenberg, Chair of the Advisory Committee

_____ _____

Dr. Gregory T. Byrd **Dr. Alexander G. Dean**

_____

Dr. Frank Mueller

BIOGRAPHY

Aravindh Venkataseshadri Anantaraman was born on 29th October 1979 in the Union Territory of Pondicherry, India. In 2001, he graduated with a B.E (Honors) degree in Electrical and Electronics Engineering from the Birla Institute of Technology and Science (BITS), Pilani, Rajasthan, India. He was an intern with Motorola GSM design center at Bangalore, India between July and December 2000.

In fall 2001, he enrolled in the masters program in computer engineering at North Carolina State University, Raleigh. He has been working under the guidance of Dr. Eric Rotenberg since spring 2002 in the area of real-time embedded systems.

Acknowledgements

First, I would like to thank my parents Anantaraman and Rukmani for their support. I cannot describe in words my gratitude for them for all they have done for me.

I would like to thank my graduate advisor, Dr. Eric Rotenberg, for having given me the opportunity to work under him. He is a fantastic teacher and I have learned a lot from him about computer architecture and about research. His enthusiasm and energy have influenced me a lot. He has shown tremendous confidence in me and I am indebted to him. His way of expressing things clearly and without ambiguity continues to amaze me. His writing style is the best that I have seen and I have tried my best to write this thesis to meet his high standards.

Sincere appreciation is due to Dr. Frank Mueller for his guidance and his valuable suggestions during the course of this project. I would also like to specially thank him for permission to use the static timing analyzer tool developed by him.

I also thank Dr. Gregory Byrd, Dr. Alexander Dean, and Dr. Frank Mueller for having agreed to be on my thesis committee and for their valuable comments after reviewing my thesis.

I would like to thank my colleague on this project Kiran Seth. It was a great experience working with him. He was fully responsible for fixing the static timing analyzer tool used in this thesis.

I would like to thank Nikhil Gupta and Prakash Ramrakhyani for having put up with my antics and for providing me with much needed breaks. They have been a great help when I needed help with the simulator. Special thanks are due to Prakash for his help with the Wattch models.

I would also like to thank the other members of my research group - Zach Purser, Karthik Sundaramoorthy, and Jinson Koppanallil - for having helped me out on various occasions. Zach's critical comments have been very thought-provoking and have helped me to look at this research from different perspectives. Special acknowledgements are due to Steve Lipa and David Winick for their help while at 301 EGRC.

My friend Vishwanath Sundararaman has been a great help. He is a great source of inspiration to me. He has helped me out in crunch times and I am thankful to him for his support.

I am deeply indebted to Lashminarayan Venkatesan and Udayakumar Shanmugam for their help in various ways.

Friends who have helped me in numerous ways include Uma Raghuraman, Prasanna Venkatesh, Balaji Sundar, Anand Natarajan, Karthik Subramanian, Narayanan Chandrasekar, Geethapriya Thamilarasu, Anupama Balasubramanian, Anand Parthasarathy, Visvanathan Hariharan, Mohamed Sheik Nainar, Jaikumaran Cancheevaram, Srivatsan Ravindran, Sajjan Raghavan, Pradeep Mahadevan, Karthikeyan Santhanagopalan, Manukaran Karunakaran, Arianathan Rajagopal, Patrick Hamilton, Yogesh Ramados, Subhashini Sivagnanam, Anita Nagarajan…the list is endless. I thank them all for their support.

I would like to take this opportunity to thank my friends from high school Varadarajan Sridharan, Denis Joe David, and Namassivayam Sandrasse. They have been a constant source of support to me.

This research has been supported by NSF grant No. CCR-0208581 and generous funding and equipment donations from Intel.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF EQUATIONS

Chapter 1 Introduction

In hard real-time systems, it is critical to ensure that a task always finishes before its deadline. Missing a deadline could have potentially hazardous effects. To guarantee safety, designers of real-time systems estimate an upper bound on the number of cycles needed for the task. The upper bound should be safe, i.e., it should always be greater than the actual execution time even in the worst possible scenario.

Static worst-case timing analysis is a means for automatically deriving a safe upper bound for the execution time. Static timing analysis performs a traversal of all possible execution paths of a program and identifies the longest execution path. More importantly, static timing analysis accounts for both program uncertainty (path uncertainty due to input dependencies, statically unknown addresses, etc.) and also microarchitectural uncertainty, (pipelining, caching, dynamic branch prediction, etc.). Unfortunately, due to the complexity of the analysis and the constraint of ensuring a safe bound, the analysis is very conservative and the upper bound on the execution cycles is typically over-inflated.

To ensure safety, the minimum clock frequency needed by the system depends on the worst-case number of cycles for a task and the task deadline.

$$frequency \geq \frac{worst\text{ - }case\ cycles}{deadline}$$

Ideally, the worst-case number of cycles should be close to the typical number of cycles, so that the frequency is close to the frequency needed in practice. Over-inflation of the worst-case bound leads to a correspondingly over-inflated frequency.

Static frequency speculation, proposed by Rotenberg [22], is a method by which a task is executed at a low speculative frequency, derived using typical execution times. Progress of the speculative task is periodically gauged. If the task is making satisfactory progress, the system continues executing the task at the speculative frequency. Otherwise, it is assumed that continued execution of the task at the speculative frequency could potentially lead to the task missing its deadline. To avoid this, the system falls back to a higher but safe recovery frequency. The recovery frequency is derived using the worst-case execution times (WCET) obtained from worst-case analysis.

Gauging progress of a speculative task is achieved by dividing it into smaller portions called sub-tasks. Each sub-task is assigned a soft interim deadline, called a checkpoint. Sub-tasks executing at the speculative frequency are expected to complete before their checkpoints, and continued safe progress of the overall task is assured as long as checkpoints are met. If a sub-task fails to complete before its checkpoint while running at the speculative frequency, the sub-task is said to have mispredicted since its actual execution time exceeds its predicted execution time. A misprediction signifies unsatisfactory progress of the task at the speculative frequency, i.e., corrective measures are needed to guarantee the overall deadline is met. Hence, the system falls back to the recovery frequency and all the remaining sub-tasks are executed at that frequency.

Reducing the system frequency by running at a low speculative frequency most of the time has benefits. Probably the most important and direct benefit is savings in power. Power is directly proportional to processor voltage and frequency by the fundamental relation $P \alpha fV^2$. Clocking the processor at a lower frequency means that circuits are permitted to run slower. Lowering supply voltage is a way of slowing down circuits. This technique of reducing frequency and hence voltage is called dynamic voltage scaling (DVS) [21]. Frequency speculation, coupled with DVS, results in cubic reductions in power. In addition, DVS contributes to savings in energy since energy varies directly as the square of voltage ($E \alpha V^2$). Thus, frequency speculation, in conjunction with DVS, enables significant savings in power and energy.

Trends in real-time systems suggest that high-performance microarchitectural techniques may be deployed to meet the ever-increasing need for performance in the realm of embedded processing. As a result, caching, branch prediction, and pipelining may soon be used in embedded processors. Although microarchitectural enhancements reduce the actual exectution time of a task, worst-case bounds for execution time will continue to be used to guarantee safety. Moreover, worst-case analysis has to account for timing complexity introduced by the microarchitectural techniques. It is difficult to foresee how well worst-case analysis will scale with increasing microarchitectural complexity in terms of deriving tight bounds. However, we expect bounding execution time of complex tasks on highly dynamic substrates will be even more difficult than bounding execution time on simple pipelines, widening the gap between worst-case

execution time and actual execution time. Figure 1-1 illustrates this trend qualitatively. Increasing microarchitectural complexity is shown on the X-axis and execution time on the Y-axis. While it is true that typical execution time scales down as microarchitectural complexity increases, the statically-derivable worst-case execution time may decrease at a slower rate, stay roughly the same, or even increase. However, irrespective of the scaling of WCET, reduction in actual execution time (brought about by the above mentioned microarchitectural techniques) will most likely increase the gap between worst-case execution time and typical execution time as shown in the figure. Since frequency speculation efficiently reconciles this gap, it is especially important that frequency speculation be employed in future embedded processors.

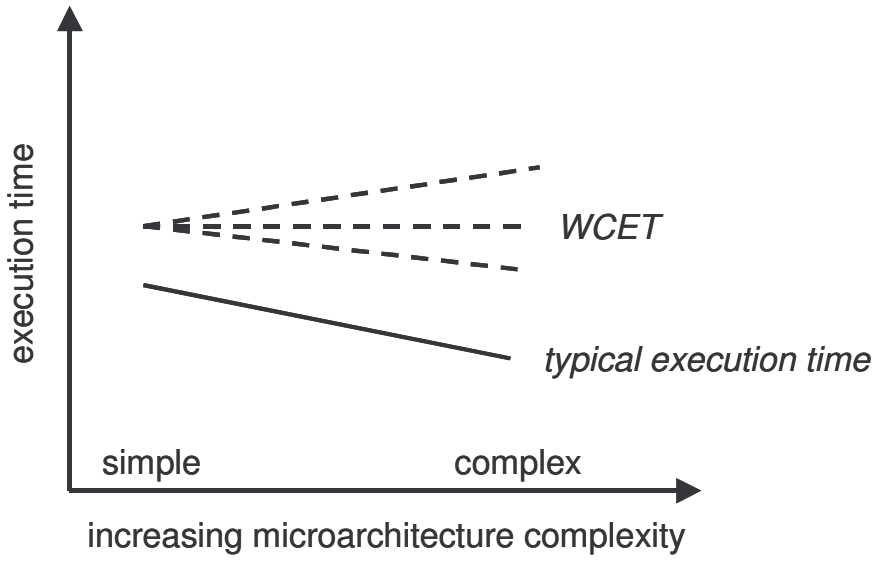


Figure 1-1. Effect of microarchitectural techniques on actual and worst-case execution times.

1.1 Contributions

The contributions of this thesis are as follows.

- Two new static frequency speculation algorithms have been proposed in this thesis that significantly outperform the original static frequency speculation algorithm [22]. The mathematical models for these algorithms have been derived. The key innovation is detecting mispredictions early, by means of a watchdog timer, namely at the checkpoint instead of at sub-task completion. This modification further decouples the speculative frequency from the worst-case execution time.

- Hardware and software requirements for static frequency speculation have been identified and described in detail.

- Sub-task selection is essential for implementing the proposed frequency speculation algorithms. Rotenberg's initial published work implemented sub-task selection by concatenating sixteen FFT programs to form the hard-real time task, each FFT program representing a sub-task [22]. This thesis takes a step forward and performs actual sub-task selection for six real-time benchmarks, albeit manually.

- The predicted execution times of sub-tasks play an important part in determining the speculative and recovery frequencies. It is important that the predicted execution times be as accurate as possible. Since task execution times are not

known initially, the initial predicted execution time is set to be equal to the worst-case execution time. Since the actual execution times may vary over time, the predicted execution times have to be updated periodically to reflect the actual execution times. Rotenberg proposed using off-line simulation to predict the execution times [22]. A key contribution of this thesis is an on-line method for profiling the actual execution times of the sub-tasks, and periodically updating the predicted execution times based on a target misprediction rate.

- A simulation framework, with the necessary hardware and software support for frequency speculation, has been developed during the course of this thesis. This environment accurately models a real-time system employing frequency speculation, including the processor, the run-time system, and sub-task delineated tasks.

- The Wattch power models [3] have been adapted to the above mentioned simulation framework. This has enabled measurement of the power consumption by a real-time system. The ability to measure power consumption in the simulation environment has enabled quantifying the savings in power due to frequency speculation techniques, as opposed to only measuring frequency reduction as done in previous work [22]. It has also enabled comparison of the different frequency speculation algorithms in terms of power savings.

1.2 Thesis Organization

Chapter 2 describes the related work that has been done in this research area. Chapter 3 focuses on the various frequency speculation algorithms and their mathematical implementation. Chapter 4 describes the hardware and software requirements of the real-time system to successfully incorporate the frequency speculation algorithms proposed in Chapter 3. The simulation environment and the benchmarks are described in Chapter 5. The results and analyses are presented in Chapter 6. Chapter 7 summarizes the conclusions of this thesis and describes future work in this research area.

Chapter 2 Related Work

This thesis is closely related to the work of Rotenberg [22] and is as an extension of that research. My work shares key aspects with the original work, including (1) exploiting the large gap between actual execution times and WCET, (2) dividing the task into smaller sub-tasks to gauge the progress, and (3) implementing static speculation, in which the speculative and recovery frequencies are computed in advance and remain fixed during the execution of the task. However, some new techniques have been developed that are unique to this thesis. One such technique is early misprediction detection, in which a misprediction is detected as early as the checkpoint via a watchdog timer, unlike the earlier method in which a misprediction is detected only at the end of the sub-task. Sub-task selection has been done manually for six real-time benchmarks and this has helped establish that it is indeed possible to do sub-task selection. In the earlier work, off-line simulations were used to estimate the predicted execution times, whereas this thesis uses an on-line profiling mechanism implemented in the real-time system. The earlier work presented savings in terms of frequency, whereas here, power and energy savings are reported using the Wattch [3] power models adapted for a contemporary superscalar microarchitecture.

Mossé, Aydin, Childers, and Melhem also proposed the idea of dividing a real-time task into sub-tasks [15]. The sub-task boundaries are called power management points (PMPs) [1]. At each PMP, frequency/voltage for remaining sub-tasks is adjusted based on the time elapsed to that point. As such, the frequency scaling algorithm aims to reclaim slack dynamically. In contrast, frequency speculation pre-computes the

speculative and recovery frequencies for a task as a whole and they do not change during the lifetime of the task.

Using our static frequency speculation approach, there are at most two frequency switches during a real-time task. The first frequency switch is at the beginning of the task, when the system sets the processor frequency to the speculative frequency computed for that task. If there is a misprediction, a second switch occurs when the system falls back to the recovery frequency. A dynamic speculation scheme, like the one proposed by Mossé et al [15], may yield greater power savings by tracking small changes in the frequency demands of the task. However, the overhead of continuously monitoring the available slack, recomputing the frequency accordingly, and switching frequencies often, can be significant (especially in systems where the frequency-switching overhead is significant).

Much research has been done in using dynamic voltage/frequency scaling to reduce power consumption in general purpose computers [8][9][14][26]. The general idea is to adjust frequency to reduce power consumption by predicting future processor utilization, while maintaining performance.

Similarly, much research has been done in the area of scheduling real-time tasks on variable-frequency processors to minimize power consumption [10][11][12][13][23]. However, most techniques are based on worst-case estimates of task execution times and

work within those constraints (although some techniques exploit variations in task execution times).

Chapter 3 Frequency Speculation Algorithms

Safe planning in real-time systems requires having upper bounds for the execution times of tasks. The worst-case execution time (WCET) is statically derived either manually by the designer or automatically by a timing analyzer integrated with the compiler. Correct worst-case timing analysis ensures that WCET is never underestimated. At the same time, overestimation of WCET is minimized as much as possible. However, as microarchitectural complexity increases, WCET may become increasingly exaggerated due to analysis complexity. Since WCET is used as a basis for computing the lower frequency bound of the real-time system, the frequency needed to guarantee a safe system is also highly inflated.

Static frequency speculation reconciles the large gap between worst-case execution time and typical execution times. The real-time task is divided into smaller sub-tasks. Typical execution times of the sub-tasks are obtained from off-line simulation, and WCET of the sub-tasks are obtained from static worst-case timing analysis. The typical execution times provide tight but unsafe bounds for the number of cycles for sub-tasks. These bounds serve as the basis for a low speculative frequency. The overall deadline is met at the speculative frequency as long as the sub-tasks finish within their predicted bounds. If a sub-task exceeds its predicted bound, then the system switches to a higher recovery frequency. Executing remaining sub-tasks at the recovery frequency guarantees that the task will finish before the deadline, in spite of the single mispredicted sub-task.

Static frequency speculation has been shown to significantly reduce frequency while maintaining system safety.

3.1 Frequency Speculation Overview

As previously mentioned, the hard real-time task is divided into smaller portions called sub-tasks. The process of splitting up the task into sub-tasks is called sub-task selection. Sub-task selection can be performed manually by the programmer or automatically by the compiler.

Each sub-task has its own interim deadline, called a checkpoint. A sub-task is expected to complete before its checkpoint at the speculative frequency. Continued safe progress of the task as a whole is confirmed by successfully completing a sub-task before its checkpoint. Note that checkpoints are soft deadlines and their purpose is only to gauge progress at the speculative frequency. Hence, a checkpoint can be missed without compromising overall safety. Recovery ensures that the overall task deadline will still be met in spite of missing a checkpoint.

Missed checkpoints are detected by means of a hardware watchdog timer. The setting of the checkpoints and the operation of the watchdog timer will be described in detail in Chapter 4.

If a checkpoint is missed, the system stops speculation and falls back to the recovery frequency, since further operation at the speculative frequency could cause the overall deadline to be missed. Sub-tasks after the mispredicted sub-task are executed at

the recovery frequency. How the mispredicted sub-task itself is handled depends on the frequency speculation algorithm, as we explain below. The mispredicted sub-task is special in that it is unfinished when the watchdog timer expires.

Figure 3-1 shows the timing of a task in which all the sub-tasks are correctly predicted (and hence has no mispredictions). The entire task is run at the speculative frequency since there are no mispredictions. Figure 3-2 and Figure 3-3 show the timing of a task in which one of the sub-tasks is mispredicted. There is a key difference in the method for detecting the misprediction, as illustrated in the two figures. In the late-detection model (Figure 3-2), the value in the watchdog timer is examined on the completion of the sub-task. If the value is zero (expired), it means that the actual execution time of the sub-task is more than its predicted execution time. Thus, a misprediction must have occurred, so remaining sub-tasks are executed at the recovery frequency.

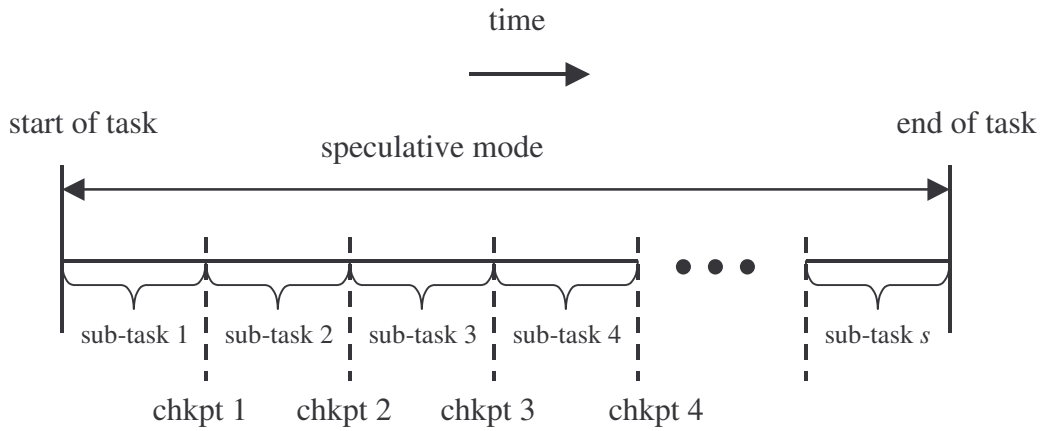


Figure 3-1. Timing of a task with no mispredictions.

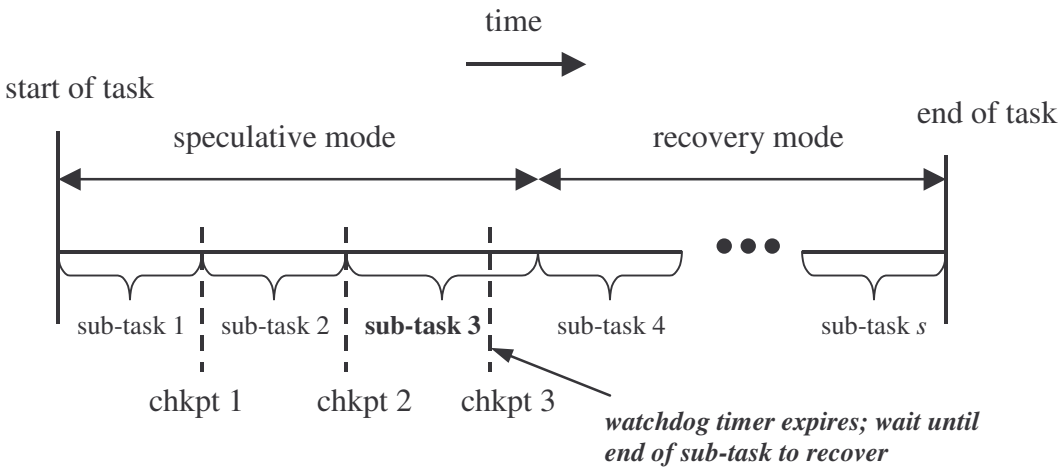


Figure 3-2. Timing of a task with one misprediction and late-detection.

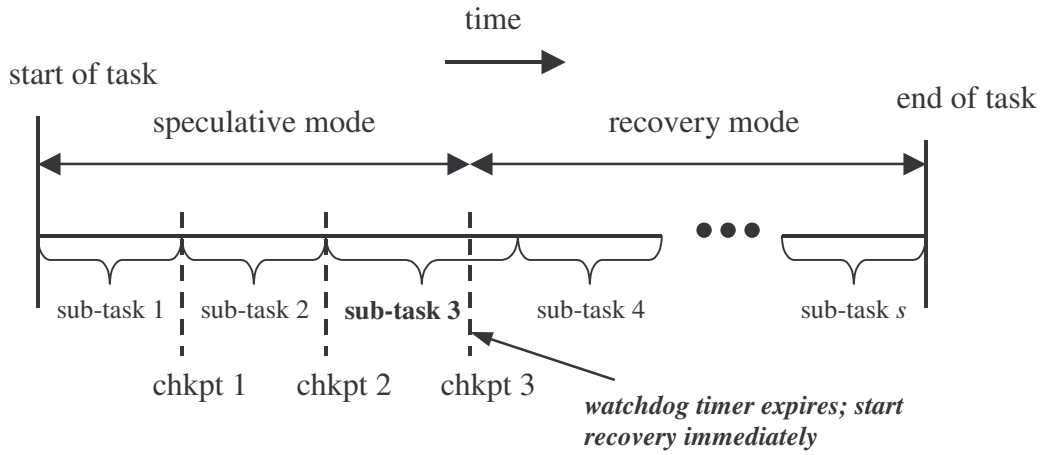


Figure 3-3. Timing of a task with one misprediction and early-detection.

The late detection mechanism has a drawback, however. Recovery is delayed since a misprediction is detected only at the end of the sub-task. If the misprediction were detected earlier, recovery could be initiated earlier and the unfinished portion of the mispredicted sub-task could be completed at the higher recovery frequency. We know that missing a checkpoint is the earliest indication of a misprediction. Detecting a misprediction at the checkpoint itself provides a means to execute the unfinished portion

of the mispredicted sub-task at the recovery frequency. In the early detection method (Figure 3-3), the watchdog timer raises an interrupt as soon as it expires (actual execution time exceeds predicted execution time).

Figure 3-4 shows the advantage of early detection over late detection. Two identical tasks with the same mispredicted sub-task (highlighted) are shown. The first case employs late detection, while the second employs early detection. The correctly predicted sub-tasks (prior to the mispredicted sub-task) and the subsequent sub-tasks (after the mispredicted sub-task) are not explicitly delineated. The amount of work in the correctly predicted sub-tasks (say X cycles) is the same in both cases. For the same frequency f_{spec}, the time spent is equal to X/f_{spec} (shown as T_1 in the figure) for both cases. Similarly, the amount of work for the sub-tasks after the mispredicted sub-task (say W cycles) is the same in both cases and hence the amount of time needed for the remaining sub-tasks is W/f_{rec} (shown as T_4 in the figure) for both cases, for the same recovery frequency f_{rec}. Consider now the amount of time needed for the mispredicted sub-task. The amount of work completed before the checkpoint (say Y cycles) is the same since both cases have the same speculative frequency. The time taken is Y/f_{spec} (shown as T_2 in the figure). Let Z be the amount of work (in cycles) remaining after the checkpoint. In the first case, the remaining work (Z cycles) is executed at the speculative frequency ($T_3 = Z/f_{spec}$), whereas in the second case, the unfinished work is executed at the higher recovery frequency ($T_3' = Z/f_{rec}$). The portion of the mispredicted sub-task after the checkpoint is executed faster in the second case because it is executed at a higher frequency, i.e., $T_3' < T_3$ in the figure. The total time to execute the entire task is less for

the early detection method than for the late detection method as shown in the figure, assuming the same $\{f_{spec}, f_{rec}\}$ pair. The extra slack in the timeline generated by initiating recovery earlier can be utilized to *lower* the speculative frequency, in case of early detection.

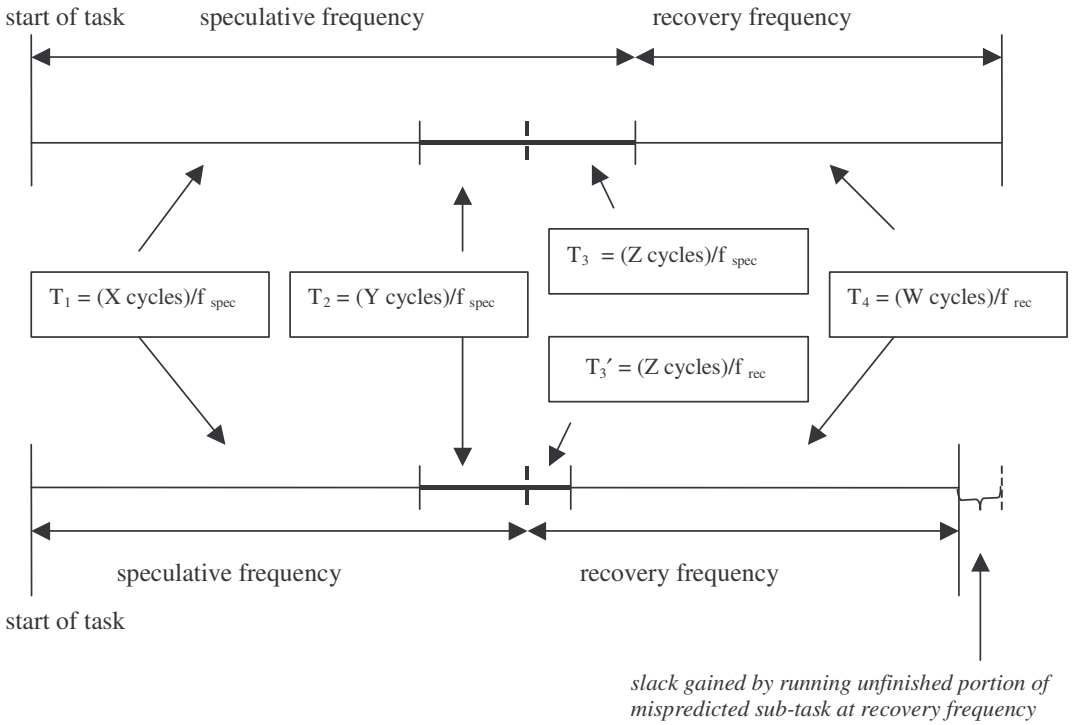


Figure 3-4. Advantage of early detection over late detection.

Above, we explained how early detection has the potential to yield a lower speculative frequency. However, a new problem arises, in particular determining a tight yet safe bound for the unfinished work of a mispredicted sub-task (Z cycles in Figure 3-4). The simplest solution is to bound the unfinished work using the worst-case number of cycles for the *entire* sub-task. This is tantamount to re-executing the mispredicted sub-task. In reality, the sub-task is not re-executed. The notion of "re-execution" is only an abstraction for simplifying the mathematical analysis. Hence this method for bounding the unfinished work is called logical re-execution. Logical re-execution is highly

pessimistic since it assumes no work is completed by the mispredicted sub-task at the speculative frequency (before the misprediction is detected). Nevertheless, as long as the time spent by the mispredicted sub-task until the checkpoint plus the time needed to logically re-execute the sub-task (at the high recovery frequency assuming worst-case conditions) is less than the time needed to execute the entire sub-task at the low speculative frequency assuming worst-case conditions, the early-detection logical re-execution method will outperform late-detection. As we will show, this is often the case.

However, the penalty of logical re-execution is significant and reduces potential frequency savings due to early-detection. To overcome this, a tighter bound for the unfinished work and hence the time needed to complete the unfinished work at the recovery frequency is derived using a more sophisticated analysis. This tighter method, called early-detection continuous-execution, effectively addresses the problems of both late-detection and early-detection with logical re-execution.

3.2 Terminology

The notation that will be used throughout this thesis (to describe the characteristics of the system that uses frequency speculation) is described in this section.

- The total number of sub-tasks in the hard real-time task is denoted by the letter s.
- Frequency is denoted by the letter f.
- f_{wc} denotes the minimum frequency at which the processor should be run such that the deadline of the task is met, if frequency speculation is not employed. Hence f_{wc} is based solely on conventional worst-case analysis.

- f_{spec} represents the speculative frequency as determined by the frequency speculation algorithm. Typically f_{spec} is much lower than $f_{wc.}$ A system running at the speculative frequency (f_{spec}) is expected to meet the deadline, but not guaranteed to do so. Therefore, progress must be continuously gauged and a recovery mechanism deployed as needed.
- f_{rec} represents the recovery frequency as determined by the frequency speculation algorithm. The recovery frequency is the minimum frequency at which the remainder of the task must be executed to ensure that the overall deadline is met in case a sub-task is mispredicted.
- i, j, and k denote individual sub-tasks that constitute the overall task.
- Static worst-case timing analysis is performed by a static timing analyzer on a per sub-task basis. The worst-case execution time of a sub-task is denoted by $WCET_{i,f}$, where the subscript i denotes the sub-task and the subscript f denotes the frequency at which WCET is estimated.
- The predicted execution time of a sub-task i at frequency f is denoted by $PET_{i,f}$.
- The actual execution times of a sub-task i at frequency f is denoted by $AET_{i,f.}$ Actual execution time is not known until run-time and is not used directly in the analysis that generates the speculative and recovery frequencies.

$WCET_{i,f}$ and $PET_{i,f}$ are the two key inputs for the frequency speculation algorithms described in Section 3.3 to compute the speculative and recovery frequencies.

3.3 Frequency Speculation Algorithms

In this section, we describe the various static frequency speculation algorithms used to derive the speculative and recovery frequencies. The speculative and recovery frequencies are derived before the task executes and kept the same for the duration of the task. In this sense, frequency speculation is "static". However, a run-time software component periodically re-applies the frequency speculation algorithm to update the speculative and recovery frequencies, based on more recent history of actual execution times. (This approach is described in detail in Section 4.3.)

Before describing the frequency speculation algorithms, we define the input parameters needed for the analysis:

- Task deadline.
- Number of sub-tasks.
- Frequency range supported by the microprocessor.
- $PET_{i,f}$: Predicted execution times for each of the sub-tasks at all supported frequencies.
- $WCET_{i,f}$: Worst case execution times for each of the sub-tasks at all supported frequencies.
- Frequency switching overhead: There is a fixed penalty incurred whenever frequency is switched (e.g., from the speculative to the recovery frequency). This overhead depends on the DVS implementation of the system. In some systems, voltage and frequency are incremented/decremented in steps and the switching

overhead depends upon the number of steps between the starting and ending frequencies.

3.3.1 No speculation

The worst-case frequency f_{wc} is derived without speculation, i.e., using only worst-case execution times.

$$\sum_{i=1}^{s} WCET_{i,\,fwc} \leq deadline$$

Equation 3-1. Computing the worst-case frequency.

The worst-case frequency f_{wc} for a task, given a deadline, is computed using Equation 3-1. $WCET_{i,f}$ for all the sub-tasks are substituted into Equation 3-1 starting from the lowest available frequency and progressively increasing the frequency until the inequality is satisfied. Starting at the lowest frequency and proceeding upwards ensures that we arrive at the minimum value for f_{wc}. It is to be noted that the worst-case frequency f_{wc} is not needed for frequency speculation purposes. However, it is calculated to provide a basis for comparison.

3.3.2 Original frequency speculation algorithm

This section reviews the original frequency speculation algorithm as proposed by Rotenberg [22]. In this algorithm, the real-time task, which has been divided into sub-tasks, is initially executed at the speculative frequency. Each sub-task has an interim deadline, i.e., checkpoint. At the end of each sub-task, the system checks if that sub-task completed before its checkpoint. If it has completed before its checkpoint, the system

continues to execute the next sub-task at the speculative frequency. However, if a checkpoint is missed (actual execution time exceeds predicted execution time), then that sub-task is said to be mispredicted. On a misprediction, recovery is initiated. All subsequent sub-tasks are executed at the recovery frequency. Note that misprediction detection is late, i.e., a misprediction is detected on completion of the sub-task.

Figure 3-5 shows the timing of a task, complete with sub-tasks and checkpoints. Each speculative sub-task is expected to complete before its checkpoint, meaning its actual execution time should not exceed $PET_{i,fspec}$. At the end of each speculative sub-task, the system checks if the sub-task completed before its checkpoint. The third sub-task, marked with a cross, represents a misspredicted sub-task in Figure 3-5. The misprediction is detected at the end of the third sub-task. After a fixed switching overhead, the system falls back to the recovery mode and all the sub-tasks that follow the mispredicted sub-task are run in recovery mode.

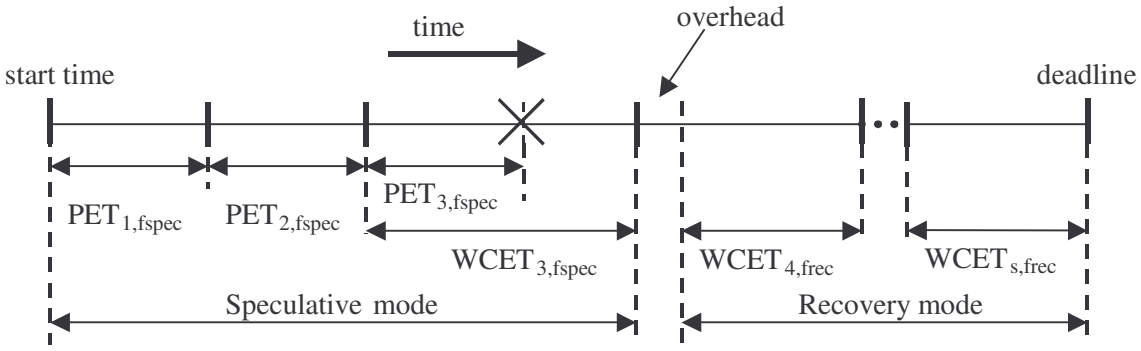


Figure 3-5. Timing of a real-time task in a system implementing the original frequency speculation algorithm.

At the end of the task, the system returns to some other speculative frequency to begin execution of the next task. There is a second switching overhead incurred in this transition from the recovery to the speculative frequency. If the task had no mispredictions, then the frequency is changed from the current speculative frequency to the speculative frequency for the next task.

Equation 3-2 is the mathematical implementation of the original static frequency speculation algorithm. The first term on the left-hand side represents an upper bound on the cumulative execution time of all the correctly predicted sub-tasks, the sum of their PETs at the speculative frequency. The second term is an upper bound on the time taken by the mispredicted sub-task, which is assumed to be the worst-case execution time of that sub-task at the speculative frequency. It is to be noted that the actual execution time of a sub-task, $AET_{i,fspec}$, is unknown until run-time. If the sub-task completes before its checkpoint, it means that $AET_{i,fspec}$ is less than or equal to $PET_{i,fspec}$. However, in the case of a misprediction, $AET_{i,fspec}$ is greater than $PET_{i,fspec}$ but less than $WCET_{i,fspec}$. In the worst case, $AET_{i,fspec}$ would be equal to $WCET_{i,fspec}$. Hence, for the purpose of safe analysis, we use WCET at the speculative frequency for the execution time of a mispredicted sub-task.

$$\sum_{j=1}^{i-1} PET_{j,\,f_{spec}} + WCET_{i,\,f_{spec}} + overhead + \sum_{k=i+1}^{s} WCET_{k,\,f_{rec}} \leq deadline$$

Equation 3-2. Mathematical representation of the original frequency speculation model.

The third term in Equation 3-2 is the fixed overhead that is incurred when switching from the speculative to the recovery frequency.

To ensure a safe system, the worst-case scenario is assumed for all the remaining sub-tasks that are executed at the recovery frequency. Note that while WCET is assumed for both the mispredicted sub-task ($WCET_{i,fspec}$) and subsequent sub-tasks ($WCET_{i,frec}$), the mispredicted sub-task is executed at the speculative frequency whereas the subsequent sub-tasks are executed at the recovery frequency. The last term in Equation 3-2 accounts for the cumulative execution time of the remaining sub-tasks assuming WCET at the recovery frequency.

The sum of the four terms on the left-hand side of Equation 3-2 must be less than or equal to the deadline specified for the real-time task to ensure a safe system.

Since any one of the s sub-tasks could be the mispredicted sub-task, Equation 3-2 actually represents s inequalities. It is essential that Equation 3-2 be satisfied for all the s sub-tasks to ensure that the deadline will be safely met. The lowest supported frequency is chosen as the starting value for the speculative frequency, and the lowest f_{rec} that satisfies the inequality in Equation 3-2 assuming the first sub-task is mispredicted is determined. It is then checked if this $\{f_{spec}, f_{rec}\}$ pair satisfies the inequality assuming the second sub-task was mispredicted, and so on for all the s sub-tasks. If there is a sub-task for which there is no f_{rec} to satisfy the inequality, the f_{spec} is incremented to the next available frequency and we iterate again through all the s inequalities until we fail to find

an f_{rec} for one of them (for that f_{spec}). This process continues until a minimum $\{f_{spec}, f_{rec}\}$ pair is found that satisfies all s inequalities.

3.3.3 Early-detection logical re-execution algorithm

As mentioned in Section 3.1, the late-detection method is not very efficient. The early-detection method was introduced to overcome the deficiencies of the late-detection method. This section describes the early-detection logical re-execution algorithm, which uses the WCET of the entire mispredicted sub-task to conservatively bound the work remaining in a sub-task after a misprediction is detected.

Figure 3-6 illustrates the timing of a task employing the logical re-execution algorithm. The first two sub-tasks are correctly predicted. The third sub-task, denoted by a cross, is mispredicted and the misprediction is detected at its checkpoint. The figure shows the entire mispredicted sub-task to be re-executed (logically). All the remaining sub-tasks are executed at the recovery frequency assuming worst-case execution times.

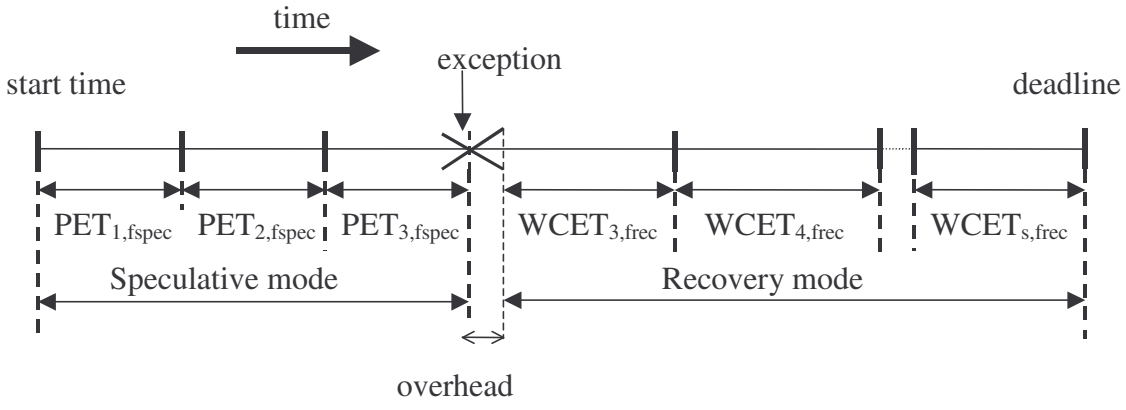


Figure 3-6. Timing of a real-time task executed on a system implementing the early-detection logical re-execution algorithm.

Equation 3-3 is the mathematical model for the logical re-execution algorithm. The first term of the inequality represents the cumulative execution time of all the correctly predicted sub-tasks. The second term represents the predicted execution time of the mispredicted sub-task at the speculative frequency. This is the amount of time that elapses before the watchdog timer raises an exception. The third term represents the overhead incurred in switching from the speculative to the recovery frequency. The fourth term very conservatively bounds the remaining time needed by the unfinished, mispredicted sub-task, at the recovery frequency. The simplest way to bound the remaining time is to use the WCET of the entire sub-task; hence it seems like the sub-task is "re-executed" (in fact, it continues normally, i.e., re-execution is apparent only in the mathematical expression). The fifth expression is the sum of the worst-case execution times of the sub-tasks after the mispredicted sub-task while running at the recovery frequency. The sum of these five terms should be less than or equal to the deadline to ensure safe operation.

$$\sum_{j=1}^{i-1} PET_{j,\,f_{spec}} + PET_{i,\,f_{spec}} + overhead + WCET_{i,\,f_{rec}} + \sum_{k=i+1}^{s} WCET_{k,\,f_{rec}} \leq deadline$$

Equation 3-3. Mathematical representation of the early-detection logical re-execution algorithm.

Again, Equation 3-3 represents s inequalities since any one of the s sub-tasks could be the mispredicted sub-task. The speculative and recovery frequencies are determined iteratively as previously described in the original frequency speculation algorithm.

The mispredicted sub-task is assumed to be *logically* re-executed at the recovery frequency only for the purposes of analysis. In reality, the mispredicted sub-task is not re-executed from the beginning of the sub-task. Only the unfinished portion of the mispredicted sub-task is actually executed at the recovery frequency. For the purpose of analysis, it is safe to assume that the worst-case execution time of the entire sub-task at the recovery frequency ($WCET_{i,frec}$) is a safe upper bound for the execution time of any part of that sub-task assuming worst-case conditions. This algorithm is pessimistic in the sense that it assumes no useful work was done by the mispredicted sub-task at the speculative frequency.

Equation 3-3 can be simplified by merging the PET and WCET terms of the mispredicted sub-task with the corresponding summation terms as shown in Equation 3-4.

$$\sum_{j=1}^{i} PET_{j,\,f_{spec}} + overhead + \sum_{k=i}^{s} WCET_{k,\,f_{rec}} \leq deadline$$

Equation 3-4. Simplified form of Equation 3-3.

3.3.4 Early-detection continuous-execution algorithm

This section describes a frequency speculation algorithm that is an optimization of the early-detection logical re-execution algorithm. In the logical re-execution algorithm, WCET of the *entire* mispredicted sub-task is used to bound the time needed to complete the unfinished portion of the sub-task. The early-detection continuous-execution algorithm derives a tighter bound for the time needed to complete the unfinished portion

of the mispredicted sub-task, thereby introducing more slack in the schedule and reducing the speculative frequency further.

In this algorithm, an exception is raised when a misprediction is detected and the unfinished portion of the mispredicted sub-task is executed at the recovery frequency. The early-detection continuous-execution algorithm estimates the amount of work that has been completed at the speculative frequency, and from this computes a tight bound for the amount of time needed to complete the unfinished portion of the mispredicted sub-task at the recovery frequency. The remaining sub-tasks are executed at the recovery frequency as in the previous algorithms.

Figure 3-7 depicts the timing of a real-time system that implements the early-detection and continuous-execution algorithm. The first two sub-tasks meet their checkpoints while the third is mispredicted. On misspeculation, the system transitions to the recovery frequency after the switching overhead and completes the mispredicted task. Subsequent sub-tasks are executed at the recovery frequency assuming worst-case conditions.

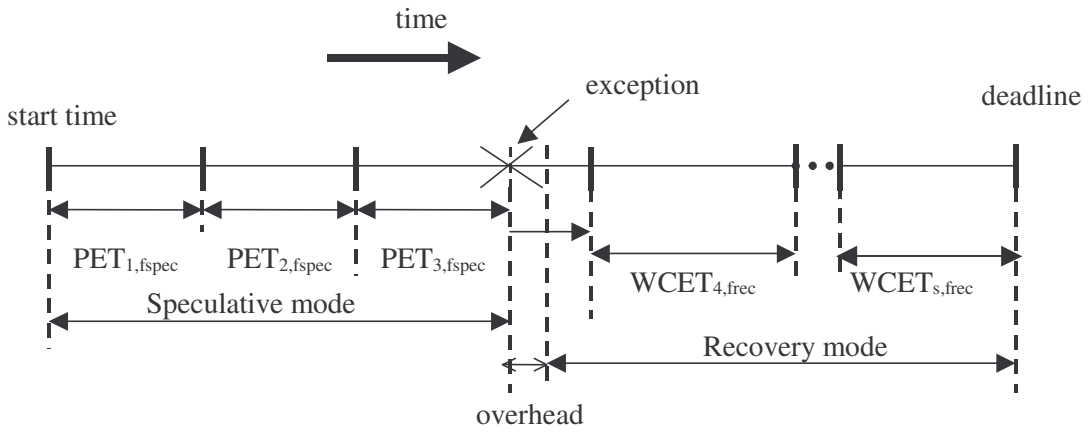


Figure 3-7. Timing of a real-time task executed on a system implementing the early-detection and continuous-execution algorithm.

Equation 3-5 is the overall equation that implements the early-detection and continuous-execution algorithm. The first term is the sum of the predicted execution times of the correctly predicted sub-tasks at the speculative frequency. The second term is the amount of time spent by the mispredicted sub-task at the speculative frequency before the exception is triggered. The third term denotes the amount of time that is needed to complete the unfinished portion of the mispredicted sub-task at the recovery frequency. The fourth term denotes the overhead incurred in switching from the speculative to the recovery frequency. The fifth term denotes the sum of the worst-case execution times of the sub-tasks following the mispredicted sub-task, executed at the recovery frequency.

$$\sum_{j=1}^{i-1} PET_{j,\,fspec} + PET_{i,\,fspec} + \text{time left to complete mispredicted sub-task} +$$

$$\text{overhead} + \sum_{k=i+1}^{s} WCET_{k,\,frec} \leq \text{deadline}$$

Equation 3-5. Mathematical equivalent of the early-detection continuous-execution algorithm.

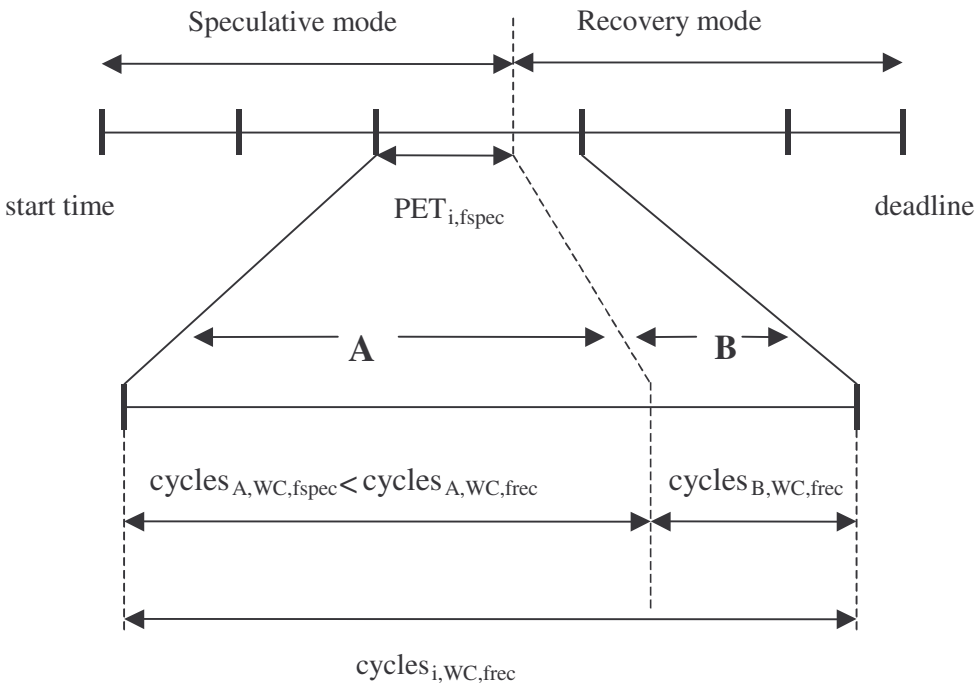


Figure 3-8. Timeline of the mispredicted sub-task in the early-detection continuous-execution scheme.

Figure 3-8 highlights the timeline of the mispredicted sub-task i. Because it was mispredicted, we must assume the worst-case scenario (WC) occurred. Hence, we bound the number of cycles for the entire sub-task using $cycles_{i,WC,f}$. Note that the number of cycles depends on frequency only due to memory effects: the number of cycles to access main memory increases with frequency. The mispredicted sub-task i is partially executed at f_{spec} and partially executed at f_{rec}. Since memory cycles increases with frequency, we can safely bound the worst-case number of cycles for sub-task i using only the higher of the two frequencies, as follows: $cycles_{i,WC,frec}$.

We divide the sub-task into two smaller components, A and B. A is the component completed before the misprediction and B is the component left to be done. The upper bound on the number of cycles derived above can be broken down as follows.

$$cycles_{i,WC,frec} = cycles_{A,WC,frec} + cycles_{B,WC,frec}$$

Equation 3-6. Number of cycles of the mispredicted sub-task at the recovery frequency.

Since we are interested in bounding the time for B, the unfinished portion, Equation 3-6 is re-arranged as follows.

$$cycles_{B,WC,frec} = cycles_{i,WC,frec} - cycles_{A,WC,frec}$$

Equation 3-7. Number of cycles that remain and must be completed at the recovery frequency.

Of course, A is executed at the speculative frequency, not the recovery frequency. We know that $cycles_{A,WC,fspec} < cycles_{A,WC,frec}$ because memory cycles increase with higher frequency. Since the term for A in Equation 3-7 is *subtracted* from the total number of cycles, it is safe to use a smaller term for A – doing so only increases the number of cycles assumed for B, the unfinished portion. So, Equation 3-7 can be safely re-expressed as follows.

$$cycles_{B,WC,frec} \leq cycles_{i,WC,frec} - cycles_{A,WC,fspec}$$

Equation 3-8.

Now, the number of cycles spent on A is simply the product of the frequency at which A was executed, f_{spec}, and the time spent executing A. The time spent executing A is the predicted execution time of sub-task i, $PET_{i,fspec}$. So we get the following expression.

$$cycles_{A,WC,fspec} = PET_{i,fspec} * f_{spec}$$

Equation 3-9.

Substituting Equation 3-9 into Equation 3-8, we get the following expression.

$$cycles_{B,WC,frec} \leq cycles_{i,WC,frec} - PET_{i,fspec} * f_{spec}$$

Equation 3-10.

The B term can be converted from cycles to time (which is what we want – a bound on the remaining *time*) by dividing it by the recovery frequency. The right-hand side of Equation 3-10 is also scaled by the reciprocal of the recovery frequency to preserve the inequality.

$$(cycles_{B,WC,frec}/f_{rec}) \leq (cycles_{i,WC,frec}/f_{rec}) - (PET_{i,fspec} * f_{spec}/f_{rec})$$

Equation 3-10 can be re-expressed as follows.

$$\text{Time left} = cyles_{B,WC,frec}/f_{rec} = WCET_{i,frec} - (f_{spec}/f_{rec})\, PET_{i,fspec}$$

Substituting the above expression into Equation 3-5, we get the final expression for the early detection and continuous execution algorithm as:

$$\sum_{j=1}^{i-1} PET_{j,\,fspec} + PET_{i,\,fspec} + WCET_{i,\,frec} - \left(\frac{f_{spec}}{f_{rec}}\right) PET_{i,\,fspec} + overhead + \sum_{k=i+1}^{s} WCET_{k,\,frec}$$

$$\leq deadline$$

Equation 3-11. Mathematical representation of the early-detection continuous-execution algorithm.

By merging the PET term of the mispredicted sub-task into the corresponding summation term, Equation 3-11 can be re-expressed as:

$$\sum_{j=1}^{i} PET_{j,\,fspec} + overhead + WCET_{i,\,frec} - \left(\frac{f_{spec}}{f_{rec}}\right) PET_{i,\,fspec} + \sum_{k=i+1}^{s} WCET_{k,\,frec}$$

$$\leq deadline$$

Equation 3-12. Simplified form of Equation 3-11.

Similar to the previous algorithms, Equation 3-12 has to be satisfied for all s sub-tasks, since any one of the s sub-tasks could be the mispredicted sub-task. The minimum $\{f_{spec,}\ f_{rec}\}$ pair that satisfies the above inequality for all s sub-tasks is determined iteratively as described before.

Chapter 4 System Requirements and Design

To implement frequency speculation in a real-time system, both software and hardware support are needed. Hardware support includes a watchdog counter, a cycle counter for measuring actual execution times of sub-tasks, and two registers that control the speculative and recovery frequency settings. Software support consists of off-line and run-time components. Off-line software support includes static worst-case timing analysis and sub-task selection. Run-time software support consists of management of hardware registers/counters at sub-task boundaries, and periodically re-calculating frequencies and checkpoints, according to the frequency speculation algorithm.

This chapter describes in detail both the hardware and software requirements and their implementation. Section 4.1 describes the hardware support. The off-line and run-time software support are discussed in Section 4.2 and Section 4.3, respectively.

4.1 Hardware Support

This section focuses on the hardware support required in a hard real-time system to safely implement frequency speculation. This includes architectural support, such as memory-mapped counters and registers, and low-level support such as multiple frequency and voltage settings.

4.1.1 Watchdog counter

The processor provides a watchdog counter. It is memory-mapped and hence accessible by software. An initial value can be stored into the watchdog counter via a store instruction. The watchdog counter contents can be read via a load instruction.

The processor decrements the watchdog counter by one every cycle. If it reaches zero, an exception is raised unless watchdog exceptions are disabled by the run-time software.

The watchdog counter is managed by software to detect missed checkpoints. Derivation of watchdog increment values is explained in Section 4.3.1.3.3 while management of the watchdog counter is described in Section 4.3.2.1.

4.1.2 Multiple frequency/voltage settings

A real-time system employing frequency speculation should be equipped with multiple frequency/voltage settings. Processors such as the Transmeta Crusoe, Intel® PXA25x series, and Intel® PXA26x series are good examples of processors with multiple voltage settings [28].

In our framework, the system can support frequencies from 100 Mhz to 1 Ghz in steps of 25 Mhz. The Intel Xscale [27] has four frequency/voltage settings and these are used as the basis for our settings. Other frequency/voltage settings have been interpolated as shown in Table 4-1.

Table 4-1. Frequency (MHz)/voltage (V) settings.

100/0.7	200/0.82	300/0.95	400/1.07	500/1.19	600/1.31	700/1.43	800/1.56	900/1.68	1000/1.8
125/0.73	225/0.85	325/0.98	425/1.1	525/1.22	625/1.34	725/1.46	825/1.59	925/1.71	
150/0.76	250/0.89	350/1.01	450/1.13	550/1.25	650/1.37	750/1.5	850/1.62	950/1.74	
175/0.79	275/0.92	375/1.04	475/1.16	575/1.28	675/1.41	775/1.53	875/1.65	975/1.77	

4.1.3 Hardware registers

This section describes additional hardware resources such as memory-mapped registers, apart from the watchdog counter, that are required to implement the frequency speculation algorithms. Specifically, the processor has to provide three extra memory-mapped registers. One of these three registers is referred to as the profiling register and the other two registers are called frequency registers.

4.1.3.1 Profiling register

The profiling register provides a means to measure the actual execution time of a sub-task. Accumulated profile information is later used in selecting the PET of a sub-task.

The profiling register is essentially a hardware cycle counter. Its contents are incremented by one every cycle by the processor. The profiling register is also memory-mapped and hence can be accessed by usual load and store instructions.

4.1.3.2 Frequency registers

The frequency registers are memory-mapped registers. Unlike the watchdog counter and the profiling register, these registers are not incremented or decremented by the hardware. These just contain the current and recovery frequency settings for the current task.

A store to the current frequency register causes the processor to switch to the specified frequency setting. It is assumed that the frequency/voltage combinations are preset in the processor and hence knowledge about the frequency is sufficient to determine the corresponding voltage setting.

A watchdog exception indicates a sub-task was mispredicted. In this case, the processor automatically copies the contents of the recovery frequency register into the current frequency register and therefore switches to that frequency. Note that watchdog exceptions are only enabled for the early-detection schemes. For the late-detection scheme, software must explicitly set the processor frequency to the recovery frequency using the current frequency register.

4.2 Off-line Software Support

4.2.1 Static worst-case timing analysis

The purpose of worst-case timing analysis is to bound the total execution time for a task irrespective of the input data. Worst-case timing analysis has to take into account two sources of execution time variability. The first is variability due to the algorithmic

properties of the task and can potentially be affected by the input data set. The second is variability due to hardware complexity, such as the pipeline, caches, and branch predictors.

Timing analysis can be performed either dynamically or statically. Dynamic timing analysis is based on simulation. This involves identifying the input data set that causes the longest path to be traversed. A brute force method of trying all possible input combinations to determine the worst-case input may not be practical. Moreover, there is no guarantee that this worst-case input would also induce worst-case behavior in terms of pipeline effects. That is, the program could exhibit worse timing behavior with a *different* set of inputs, because of interactions with hardware.

On the other hand, static timing analysis is guaranteed to derive a safe upper bound for the execution time of a task, also called worst-case execution time (WCET).

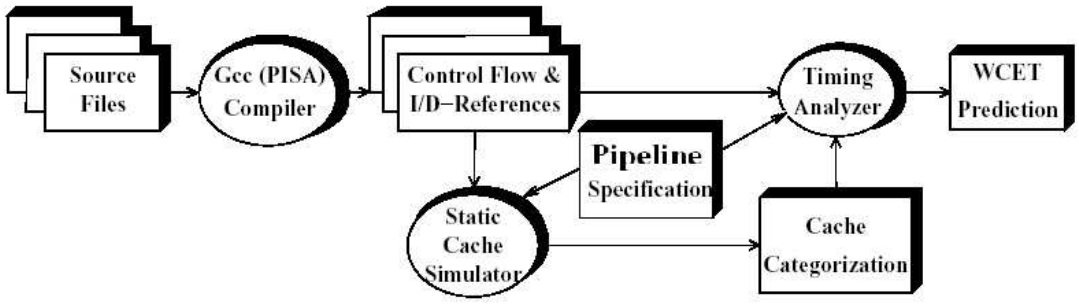


Figure 4-1. Overview of the timing tool[6][19][18][17][16][7].

Static timing analysis determines the timing information for a program by logically traversing over all potential execution paths. The analysis does not track

program values or variables. The worst-case behavior of architectural components along all execution paths is determined and a bound for the worst-case execution time in cycles is computed [2][6][7][16][17][18][20][24][25].

Figure 4-1 shows the organization of the timing tool used to statically bound WCET for an application. The complete working of the timing tool is beyond the scope of this thesis. A brief synopsis of its various components follows.

The application is compiled using the PISA gcc compiler [4]. Control flow information and references to memory are extracted from the assembly code. It is impossible to bound WCET for unbounded loops. Hence it is essential to provide the upper bound for the number of iterations for each loop.

Using the concept of *abstract cache state* [2][16], a static cache simulator generates one of four cache categorizations for each memory reference (always miss, always hit, first miss, first hit) to be used by the timing analyzer. The data cache however is not complete at this time. Hence, additional cycles have been padded onto WCET to account for actual data cache misses, obtained via simulation.

The pipeline description is an important input to the timing analysis. Since current timing analysis tools are not capable of analyzing a complex pipeline such as the one used in this thesis, we use a simplified pipeline for the purpose of timing analysis. As such, the resulting WCET is not provably safe although it is safe in practice. In fact the

resulting WCET may be too pessimistic, although with no tools currently capable of analyzing complex pipelines, this aspect cannot be assessed. To study the effect of tighter WCET, we also decrease WCET without underestimating actual execution times in practice.

The simplified pipeline model is a six-stage, scalar, in-order pipeline. Fetch, decode, register read, execute, memory access, and write-back form the pipeline stages. Only one instruction is fetched per cycle by the fetch unit. Static branch prediction is used. Forward branches are predicted not-taken, while backward branches are predicted taken. The instruction cache and the branch target buffer are merged to simplify analysis, that is, decoded branch targets are stored with the branches in the instruction cache. Instruction fetch is stalled in the case of an indirect branch, i.e., indirect branches are not predicted.

There is no register renaming since write-after-write (WAW) hazards are handled by in-order completion and write-after-read (WAR) hazards are handled by early-reads/late-writes. Read-after-write (RAW) hazards are checked by the register read unit as the values are read from the registers. A single, unpipelined function unit, with all the usual data bypasses from later pipeline stages, forms the execute stage of the pipeline. Load and store instructions access memory in program order. Thus, memory disambiguation is not needed and there is no load/store queue. Writeback is the last stage of the simplified pipeline model, during which instructions update the register file and are removed from the pipeline.

The next component of the timing tool is the timing analyzer. Using the supplied control flow information and loop bounds, cache categorizations from the static cache simulator, and the pipeline specification above, the timing analyzer bounds the timing for each path, then loops, and then functions. Thus the timing analyzer starts with the innermost loops and proceeds outwards until it reaches the outermost level. This hierarchical approach is particularly useful to us since it provides a straightforward means to query WCET for regions of the program. Thus, WCET for a specific sub-task can be easily obtained.

The tool has been used as is from its developers [6][19][18][17][16][7]. No work has been done on the tool as part of this thesis.

4.2.2 Sub-task selection

Sub-task selection is the process of dividing a real-time task into smaller parts. Sub-task selection can be done manually by the programmer or automatically by the compiler (or the static timing analyzer).

Sub-task selection has been done manually for six real-time benchmarks. Certain rules of thumb have been followed while manually selecting sub-tasks.

1. A sufficient number of sub-tasks are selected from the real-time task.
2. Sub-tasks are balanced in terms of execution cycles.
3. High-level control flow constructs such as loops and functions are good initial starting points for sub-task selection.

4. A key requirement is that the sub-tasks execute in a linear sequence (complex sub-task graphs are not permitted). The frequency speculation algorithms assume sub-tasks are concatenated to form the task.

Most real-time benchmarks used in this thesis are simple in structure and lend themselves easily to sub-task selection. In particular, sub-task selection has been done by peeling off chunks of iterations from the outermost loop.

4.3 Run-Time Software Support

This section describes run-time software support for frequency speculation. Run-time support can be further divided into two components. The first component is part of the run-time system (for example, the operating system) and hence separate from the task itself. This component periodically selects new PETs on the basis of latest profile information (actual execution times of sub-tasks), re-applies the frequency speculation algorithm to compute new speculative and recovery frequencies, and pre-computes new checkpoint information.

The second component is light-weight, as it is part of the real-time task itself. This component consists of code snippets inserted at sub-task boundaries. The snippets manage the various hardware registers (watchdog counter, profiling register, and frequency registers).

Figure 4-2 provides an example timeline in which task A is executed periodically. Task A has 4 sub-tasks. Light-weight code snippets at the beginning of each sub-task

manage the hardware registers. After n executions of task A, the run-time system component selects new PETs and re-computes the speculative and recovery frequencies and the checkpoints. Note that the run-time system component is not a part of task A and is executed between executions of A.

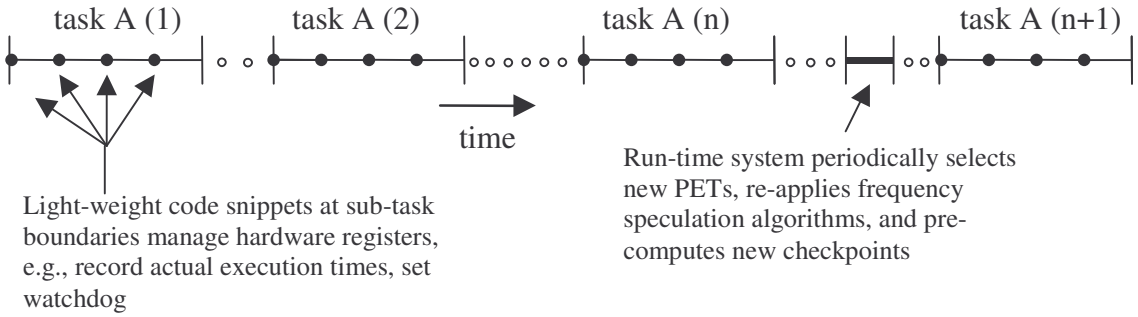


Figure 4-2. Run-time software support showing light-weight code snippets and run-time system.

4.3.1 Run-time system component

This section describes in detail the run-time system components, which includes setting new PETs, re-computing frequencies, and pre-computing the checkpoint information.

4.3.1.1 Setting PETs

Frequency speculation algorithms take advantage of the gap between WCET and PET. Experiments show that the reduction in frequency is directly correlated to the disparity between WCET and PET. A tight estimate for the predicted executed time increases the gap between PET and WCET, thereby increasing the frequency savings.

A key point is that PETs, as the name suggests, are just predictions of the execution times. The frequency speculation technique ensures safe operation even if the predicted execution times are exceeded during actual execution of the program. Hence, predicted execution times do not need to be safe bounds for execution times, and can be optimistic.

PETs that are too high, that is, which are close to WCET itself, are bad because they lead to high speculative and recovery frequencies. However, if the PETs are too low, the likelihood of mispredicting increases, reducing the power-savings potential of frequency speculation.

Predictions of sub-task execution times are made on the basis of prior actual execution times of the sub-tasks on the target processor. In prior work [22], off-line simulation was used to collect execution times of the sub-tasks and these were used as a basis for PETs. However, off-line simulation is inflexible.

A new on-line scheme to periodically update the PETs is described in this section. Actual execution times for multiple instances of a sub-task are collected by using the profiling register (details of how the profiling register is used is described in Section 4.3.2.2). These actual execution times form a history of the sub-task. Note that a separate history is maintained for each sub-task in the task. After a certain number of executions of the task, the PETs of all its sub-tasks are re-evaluated using the sub-task histories. The frequency speculation algorithm is re-run using the updated PETs, thereby calculating

new speculative and recovery frequencies. With new frequencies, the checkpoints and the watchdog increments are re-calculated also.

4.3.1.1.1 Histogram-based prediction method

In this method, the history referred to earlier is a histogram of actual execution times. Each time a sub-task executes, the actual execution time is added to its histogram. Periodically, the run-time system examines the histograms and updates the PETs for all sub-tasks based on a pre-defined target misprediction rate. The misprediction rate essentially represents the rate of sub-task mispredictions that can be tolerated by the system. A misprediction rate of zero means that the system has zero tolerance towards mispredictions and the PET is set at the highest non-zero bin. A higher misprediction rate would mean that the PETs can be more optimistic and it is possible that a sub-task might be mispredicted because the PETs might be lower than future actual execution times.

Figure 4-3 shows a hypothetical distribution of actual execution times. The X-axis represents the bins for the actual execution times and the Y-axis represents the number of samples. The distribution is assumed to be a Gaussian distribution. Assume that the total number of task executions and hence the total number of sample points is 100. The dashed line indicates a misprediction rate of 0.1. The PET is chosen to be the value at which this line intersects the X-axis. In doing so, actual execution time exceeds the predicted execution time 10% of the time and hence there is a 10% probability of the sub-task exceeding its PET, leading to a misprediction. The misprediction rate is left as a design choice for the system designer.

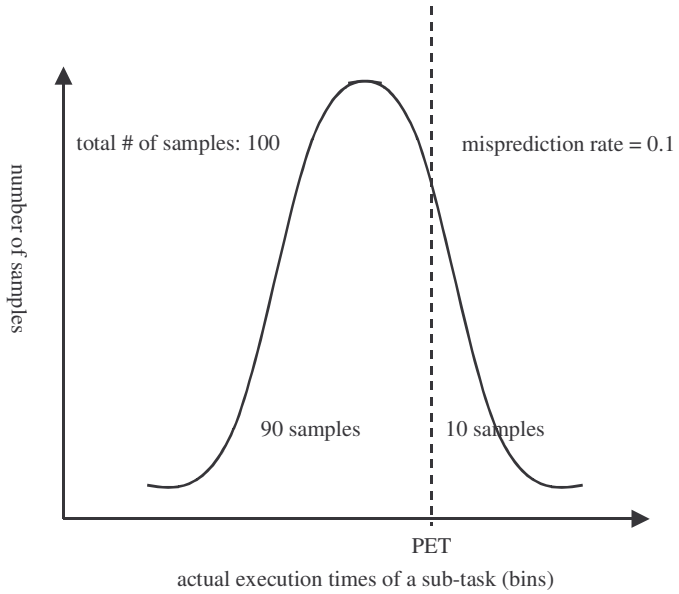


Figure 4-3. Setting PET on the basis of a target misprediction rate.

An interesting point to consider while setting the misprediction rate is the usage of different misprediction rates for different sub-tasks, since separate histograms are maintained for each sub-task. For example, if the first sub-task is mispredicted, then almost the entire task has to be executed at the recovery frequency. So, a low misprediction rate could be used for earlier sub-tasks. However, the last sub-task mispredicting might not be expensive in terms of power consumption. Also, later sub-tasks can typically utilize slack generated by previous sub-tasks and the chances of mispredicting are fairly low. In such cases, a higher misprediction rate could be used for those sub-tasks.

4.3.1.1.2 Last-N-based prediction method

In the last-N-based prediction method, the last N actual execution times are recorded and the maximum of these is chosen to be the PET.

4.3.1.2 Recomputing frequencies

After setting new PETs for the sub-tasks of a task, the run-time system component re-applies the frequency speculation algorithm described in Section 3.3, and thereby re-computes the speculative and recovery frequencies for the next N executions of the task.

4.3.1.3 Pre-computing checkpoint information

Checkpoints provide a mechanism to monitor progress of speculative sub-tasks and are critical for safe system operation. Checkpoints are pre-computed by the run-time system component. There are two methods of setting checkpoints: forward checkpointing and backward checkpointing.

4.3.1.3.1 Forward checkpointing

In the forward checkpointing methodology, the checkpoint for a sub-task is the accumulated predicted execution time of the sub-task and all prior sub-tasks at the speculative frequency, as shown in Equation 4-1.

$$\text{checkpoint}_i = \sum_{j=1}^{i} \text{PET}_{j,\text{fspec}}$$

Equation 4-1. Checkpoint for sub-task i computed using the forward checkpointing method.

4.3.1.3.2 Backward checkpointing

In the backward checkpointing methodology, we proceed backwards from the deadline to calculate the checkpoints. If a sub-task is mispredicted, there must be enough time between then and the deadline to (1) switch the frequency, (2) execute the remainder

of the mispredicted sub-task, and (3) execute all subsequent sub-tasks at the recovery frequency. As derived in Section 3.3.3, the time needed to complete the unfinished portion of the mispredicted sub-task can be conservatively bounded by the worst-case execution time of the entire sub-task. The time taken to execute the remaining sub-tasks is bounded by the cumulative worst-case execution times of those sub-tasks. The time taken to switch frequency is a fixed overhead. Hence, the checkpoint for sub-task i is as follows.

$$\text{checkpoint}_i = \left(\text{deadline} - \text{overhead} - \sum_{k=i}^{s} \text{WCET}_{k,\text{frec}} \right)$$

Equation 4-2. Checkpoint for sub-task i using the backward checkpointing method, assuming early-detection logical re-execution algorithm.

Equation 4-2 shows the checkpoint for sub-task i relative to the beginning of the task. Assuming that the total number of sub-tasks is s, the difference between the checkpoint and the deadline is enough to accommodate the frequency switching overhead and the time to execute the entire mispredicted sub-task i and all subsequent sub-tasks at the recovery frequency assuming worst-case conditions.

With more sophisticated analysis (for early-detection continuous-execution), we were able to bound the remaining time of the mispredicted sub-task with a more complex expression and hence the checkpoint for sub-task i is as follows.

$$\text{checkpoint}_i = \left(\text{deadline} - \text{overhead} - \sum_{k=i}^{s} \text{WCET}_{k,frec} + \left(\frac{f_{spec}}{f_{rec}} \right) \text{PET}_{i,fspec} \right)$$

Equation 4-3. Checkpoint for sub-task i using the backward checkpointing method, assuming early-detection continuous-execution algorithm.

4.3.1.3.3 Watchdog increment amounts

The checkpoints are transformed into watchdog increment amounts, for simple management of the watchdog counter. The first sub-task initializes the watchdog counter to the number of cycles between the start of the task and the first checkpoint, which is equal to ($checkpoint_1 * f_{spec}$). Each new sub-task adds to the watchdog counter the number of cycles between the previous checkpoint and its checkpoint. That is, a sub-task i adds (($checkpoint_i - checkpoint_{i-1}$) * f_{spec}) cycles to the watchdog counter. As long as sub-tasks finish before their checkpoints, the watchdog counter will not reach zero (mispredict) before it is incremented by the next sub-task.

Note that by augmenting the watchdog counter instead of overwriting it, extra cycles may accumulate due to sub-tasks that complete ahead of schedule. This slack provides extra time for later sub-tasks. Accumulated slack can lead to the situation where a sub-task exceeds its PET, yet completes before its checkpoint.

4.3.2 Management of hardware registers

The second component of run-time software support is part of the real-time task itself. It involves simple management of the hardware counters such as the watchdog counter, the profiling counter, and the frequency registers.

4.3.2.1 Management of the watchdog counter

A code snippet at the beginning of the first sub-task initializes the watchdog counter, to the number of cycles between the start of the task and the first checkpoint. A code snippet at the beginning of every subsequent sub-task increments the watchdog counter, using the watchdog increment amounts pre-computed by the run-time system component.

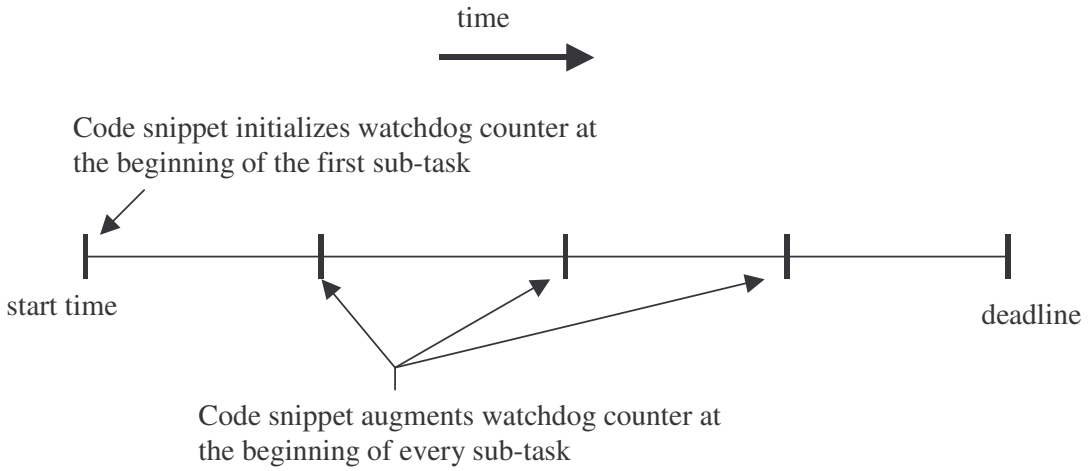


Figure 4-4. Maintenance of watchdog counter by code snippets within a task.

4.3.2.2 Management of profiling counter

The profiling counter is used to count the number of cycles consumed by a sub-task. A code snippet at the beginning of every sub-task resets the value in the profiling counter to zero. The value in the profiling counter is incremented every cycle autonomously by the processor. At the end of the sub-task, another code snippet reads out the value currently in the profiling counter. This value represents the number of cycles that were consumed by the sub-task. This value is used to generate the history for each sub-task, to be used later in setting the PETs as described in Section 4.3.1.1.

4.3.2.3 Management of the frequency registers

A code snippet at the start of the task stores the pre-computed speculative and recovery frequencies into the current frequency register and the recovery frequency register, respectively. In the interrupt method of misprediction detection, in case of a watchdog interrupt (misprediction), the processor copies the contents of the recovery frequency register into the current frequency register, causing the system to switch to the recovery frequency. However, if mispredictions are checked at the end of sub-tasks, i.e., watchdog interrupts are disabled, then a code snippet stores the recovery frequency into the current frequency register, causing the system to switch to the recovery frequency.

Chapter 5 Experimental Framework

5.1 Simulator Description

A detailed cycle-accurate simulator forms the basis for the simulation environment. The Simplescalar ISA (PISA) [4] has been used. The simulator models a dynamically scheduled 4-issue processor with 128-entry reorder buffer, 64-entry issue queue, 64-entry load/store queue, 4 pipelined universal functional units, and 2 ports to both the load/store queue and the data cache. The processor has a seven-stage pipeline. The seven stages are the fetch, dispatch, issue, register read, execute/memory, write-back, and retire stages. Conditional branches are predicted using a 2^{16}-entry *gshare* predictor. A 2^{16}-entry table indexed like the *gshare* predictor is used to predict indirect branches. The instruction and data caches are 64KB 4-way set-associative each with a block size of 64 bytes. A one cycle latency is assumed on a cache hit.The microarchitecture configuration of the system is summarized in Table 5-1.

Table 5-1. Microarchitecture configuration.

Components		Parameters
Caches	L1 instruction cache	64KB, 4-way set-assoc., 64B block, 1 cycle hit
	L1 data cache	64KB, 4-way set-assoc., 64B block, 1 cycle hit
Branch predictor	2^{16}-entry *gshare* predictor for conditional branches	
	2^{16}-entry table indexed like *gshare* for indirect branches	
Superscalar core	dispatch/issue/retire width	4 per cycle
	reorder buffer	128 entries
	issue queue	64 entries
	load/store queue	64 entries
	function units	4 pipelined, universal
	number of ports to load/store queue and D$	2 each
Execution latencies	MIPS R10K latencies	

Table 5-2 shows the supported frequencies and the corresponding memory latencies in cycles. Note that these memory latencies might be exceeded because of memory system contention due to prior outstanding cache misses.

Table 5-2. Frequency (Mhz) /memory latency (cycles) .

100/10	200/20	300/30	400/40	500/50	600/60	700/70	800/80	900/90	1000/100
125/13	225/23	325/33	425/43	525/53	625/63	725/73	825/83	925/93	
150/15	250/25	350/35	450/45	550/55	650/65	750/75	850/85	950/95	
175/18	275/28	375/38	475/48	575/58	675/68	775/78	875/88	975/98	

5.2 Power Modeling

Power modeling is necessary to measure the power and energy consumed. The Wattch power models [3], developed at Princeton, have been integrated into the simulator for this purpose. The Wattch models are based on the Register Update Unit (RUU) microarchitecture. Modifications were made to model structures in a contemporary superscalar processor, such as a separate physical register file, active list, issue queue, and load/store queue. The interested reader is referred to the paper by David Brooks et al. [3] for details about the implementation of the power models.

Support for dynamic voltage scaling has been added. The power models use voltage/frequency settings loosely based on the Intel Xscale settings [27]. Table 4-1 (Page 35) lists the voltage/frequency pairs used.

Perfect clock gating, with and without 10% standby power, is used in the experiments [3]. Standby power accounts for power dissipated in an otherwise idle unit.

The overheads of various software components of frequency speculation are included in the power measurements (managing the watchdog counter, profiling the actual execution time, re-evaluating the predicted execution times, re-computing the speculative and recovery frequencies, re-computing checkpoint information).

5.3 Benchmarks

Six benchmarks have been downloaded from the C-lab real-time benchmark suite [29]. This selection represents a majority of the contributors (FSU, Gothenburg, and Seoul National University) and represents a mix of DSP, matrix-oriented, and integer benchmarks.

Table 5-3 presents the six benchmarks and their characteristics. In all experiments, a task is run 200 times consecutively as if it were a periodic task.

Table 5-3. Benchmarks and characteristics.

name	#of sub-tasks	#of dynamic instr./task	deadlines(ms) (tight/loose)	WCET (μs)	initial PET (μs)	steady state PET (μs)	WCET/ initial PET	WCET/ steady state PET
adpcm	8	1,985,842	3.5/5.1	3286	1600	628	2.05	5.23
cnt	5	70,615	0.082/0.12	72	50	21	1.44	3.42
fft	10	219,243	0.46/0.66	426	275	72	1.54	5.91
lms	10	112,803	0.19/0.29	173	125	40	1.38	3.12
mm	10	1,652,980	2.2/3.4	2056	1620	650	1.26	3.16
srt	10	1,626,763	3.8/5.8	3508	2800	525	1.25	6.68

Chapter 6 Results

6.1 Frequency Reduction

Savings achieved due to the frequency speculation algorithms is measured in terms of frequency reduction and in terms of power. This section presents the savings achieved in terms of frequency reduction.

6.1.1 Varying deadlines

The speculative frequencies generated by the three frequency speculation algorithms are shown in Figure 6-1(a) through Figure 6-6(a) for the six benchmarks. The recovery frequencies are shown in Figure 6-1(b) through Figure 6-6(b). The deadline (in milliseconds) is varied on the X-axis, starting from a loose deadline and decreasing the deadline to the tightest deadline. Frequency is shown on the Y-axis. The available frequencies range from 100 Mhz to 1000 Mhz in 25 Mhz increments. The worst-case frequency is the frequency computed using the worst-case execution times, with no frequency speculation. The worst-case execution times were generated by the timing analyzer tool. For these experiments, the PETs are the actual execution times for the last instance of the task (i.e., the 200^{th} instance). This ensures a sufficient training period, i.e., the PETs have stabilized and reached a steady state.

The first observation is that there is a considerable reduction in the frequency between the speculative frequency and the worst-case frequency for all three frequency speculation algorithms and across all benchmarks. The speculative frequency slowly

increases as the deadline becomes tighter. This is expected since a higher speculative frequency is needed to meet a tighter deadline.

While the performance of all three algorithms is impressive, the original frequency speculation algorithm provides the least frequency reduction. The early-detection continuous-execution algorithm provides the most savings, followed by the early-detection re-execution algorithm. The reason is that the original frequency speculation algorithm assumes that the mispredicted sub-task is executed entirely at the speculative frequency, whereas the other two algorithms do not. It is to be noted that although the full sub-task is logically re-executed in the early-detection re-execution algorithm, the logical re-execution is carried out at a much higher frequency, the recovery frequency. As expected, early-detection continuous-execution is observed to be the best of the three algorithms. For example, for adpcm, at a deadline of 3.42 ms, the worst-case frequency is 975 Mhz. The speculative frequencies are 775, 675, and 400 Mhz for the original frequency speculation, early-detection re-execution, and early-detection continuous-execution methods, respectively. Frequency savings (the difference between the worst-case frequency and the speculative frequency) are 200, 300, and 575 Mhz for the three methods, respectively.

The trend for the recovery frequency is that it rises and falls with constant periods in between, as the deadlines tighten. The reason for this is that the algorithm tries to minimize the speculative frequency at the expense of the recovery frequency. Once the algorithm determines that a recovery frequency cannot be found for a particular

speculative frequency, the next supported higher frequency is chosen as the speculative frequency. Since the speculative frequency is higher than before, a lower recovery frequency suffices to meet the deadline. Hence, the recovery frequency goes down at certain points. For *adpcm*, this trend is made clearer in Figure 6-7. In Figure 6-7, the worst-case, speculative, and recovery frequencies are plotted for varying deadlines, for the original frequency speculation method. For example, at a deadline of 4.95 ms, the speculative frequency is 200Mhz and the recovery frequency is 1000Mhz. At the next lesser deadline (4.92 ms), the speculative frequency is seen to go up to 225Mhz, while the recovery frequency goes down from 1000Mhz to 950Mhz. Figure 6-8 and Figure 6-9 plot the worst-case, speculative, and recovery frequencies for *adpcm* for the other two algorithms and a similar trend is observed in those algorithms too.

The behavior of the speculative and recovery frequencies suggests an interesting trade-off. It is seen that the algorithms try to optimize the speculative frequency at the expense of the recovery frequency. This can be disadvantageous for tasks that show a high tendency for mispredictions since a high recovery frequency means more power consumption. Sometimes, increasing the speculative frequency to the next available frequency helps reduce the recovery frequency and this can be used advantageously, in terms of both reducing the misprediction rate and reducing the power consumed in recovery mode.

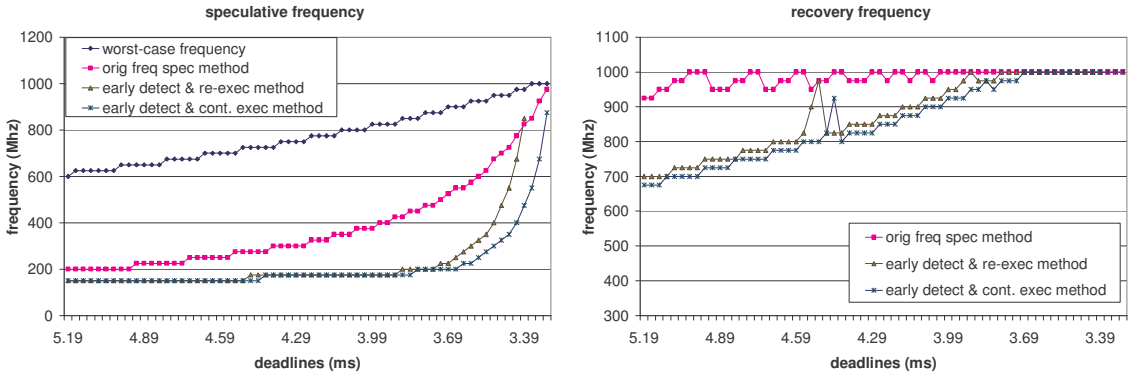


Figure 6-1. (a) Speculative and (b) recovery frequencies generated by different frequency speculation algorithms for decreasing deadlines for *'adpcm'*.

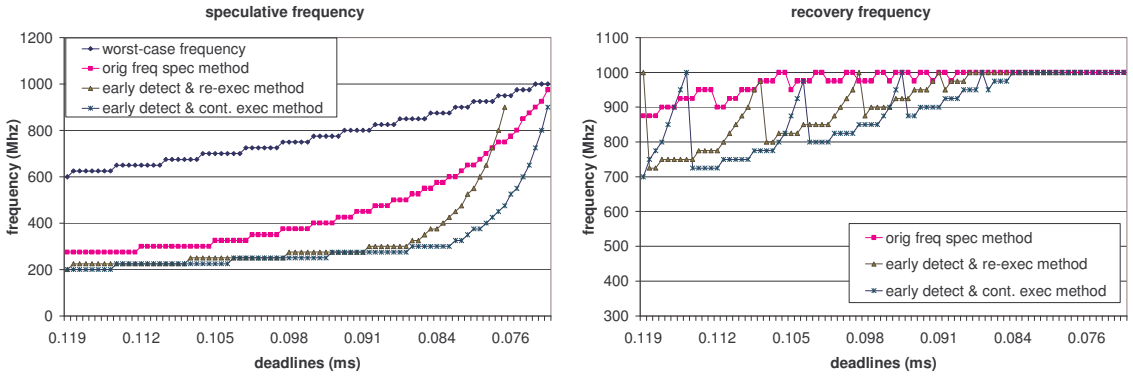


Figure 6-2. (a) Speculative and (b) recovery frequencies generated by different frequency speculation algorithms for decreasing deadlines for *'cnt'*.

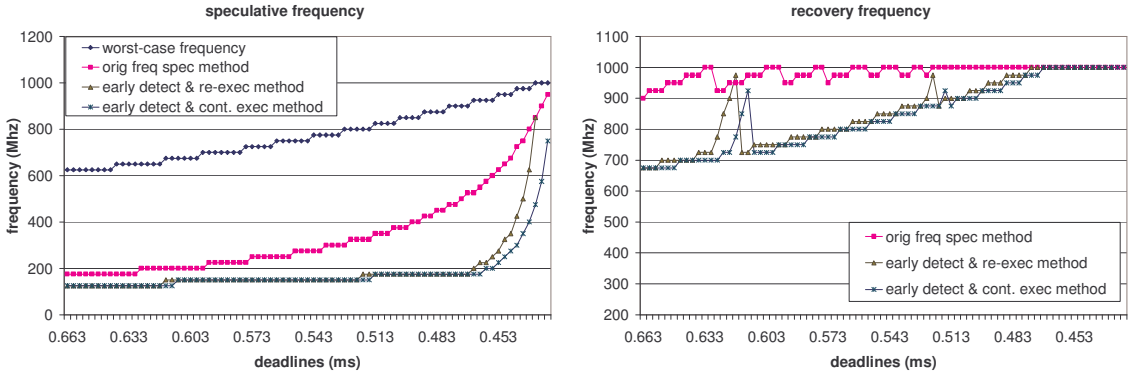


Figure 6-3. (a) Speculative and (b) recovery frequencies generated by different frequency speculation algorithms for decreasing deadlines for *'fft'*.

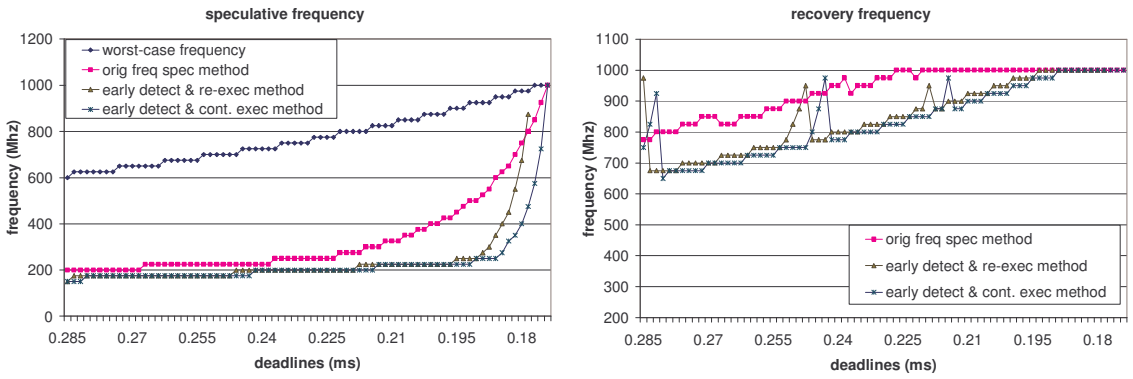


Figure 6-4. (a) Speculative and (b) recovery frequencies generated by different frequency speculation algorithms for decreasing deadlines for *'lms'*.

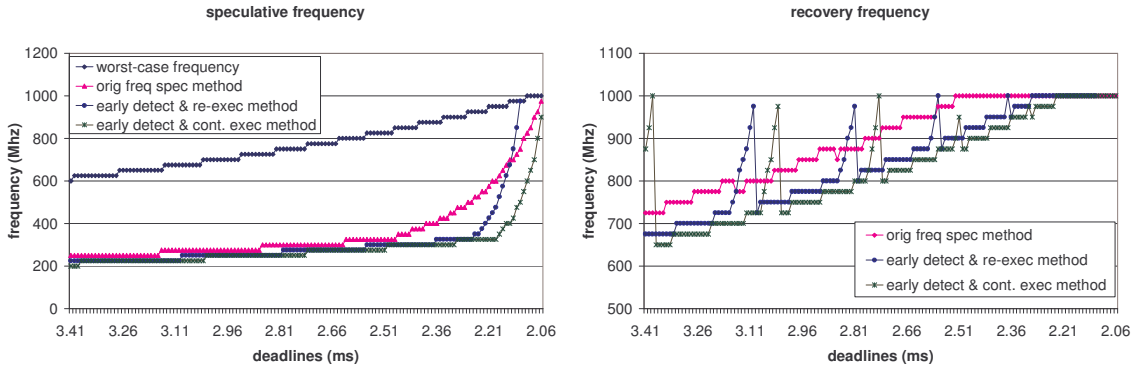


Figure 6-5. (a) Speculative and (b) recovery frequencies generated by different frequency speculation algorithms for decreasing deadlines for *'mm'*.

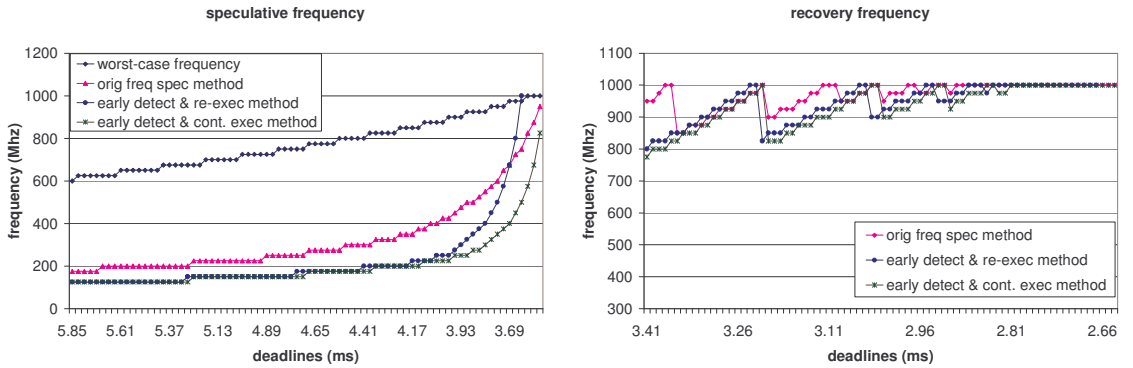


Figure 6-6. (a) Speculative and (b) recovery frequencies generated by different frequency speculation algorithms for decreasing deadlines for *'srt'*.

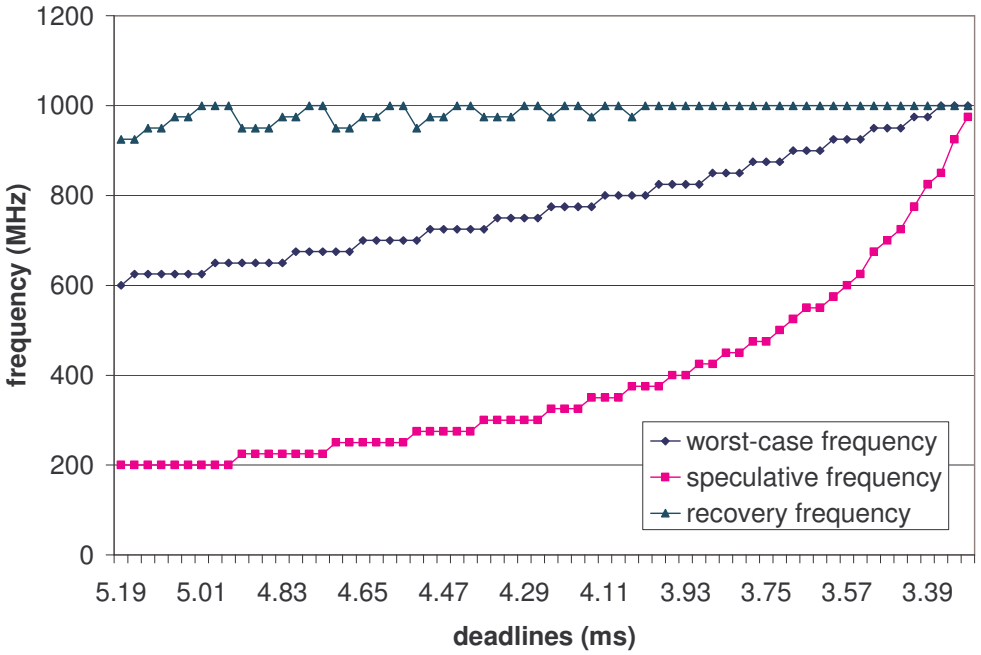


Figure 6-7. Worst-case, speculative, and recovery frequencies generated by the original frequency speculation algorithm for decreasing deadlines for *'adpcm'*.

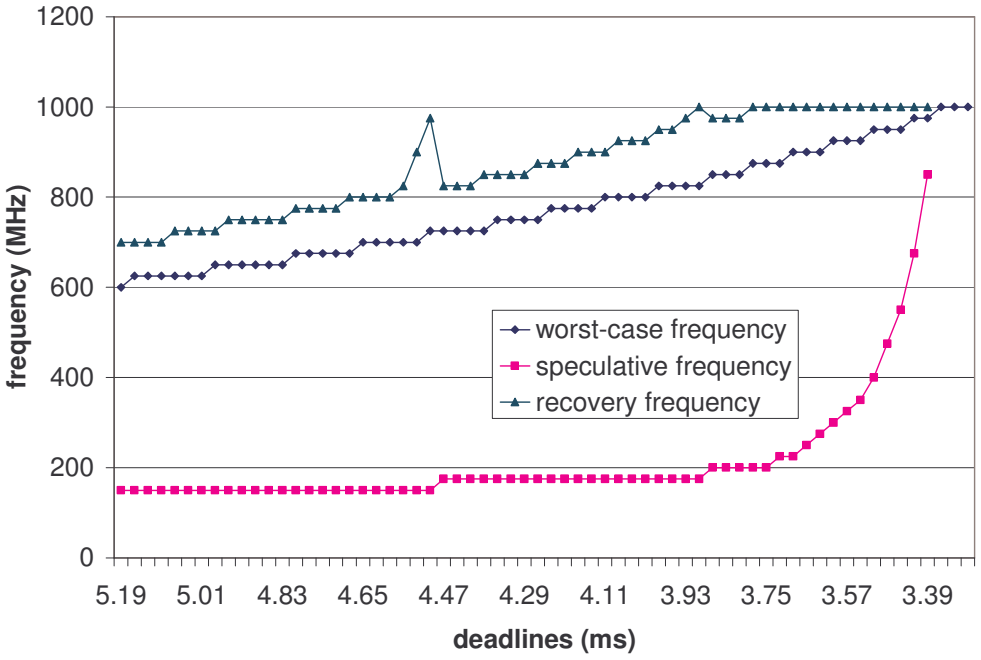


Figure 6-8. Worst-case, speculative, and recovery frequencies generated by the early-detection re-execution algorithm for decreasing deadlines for *'adpcm'*.

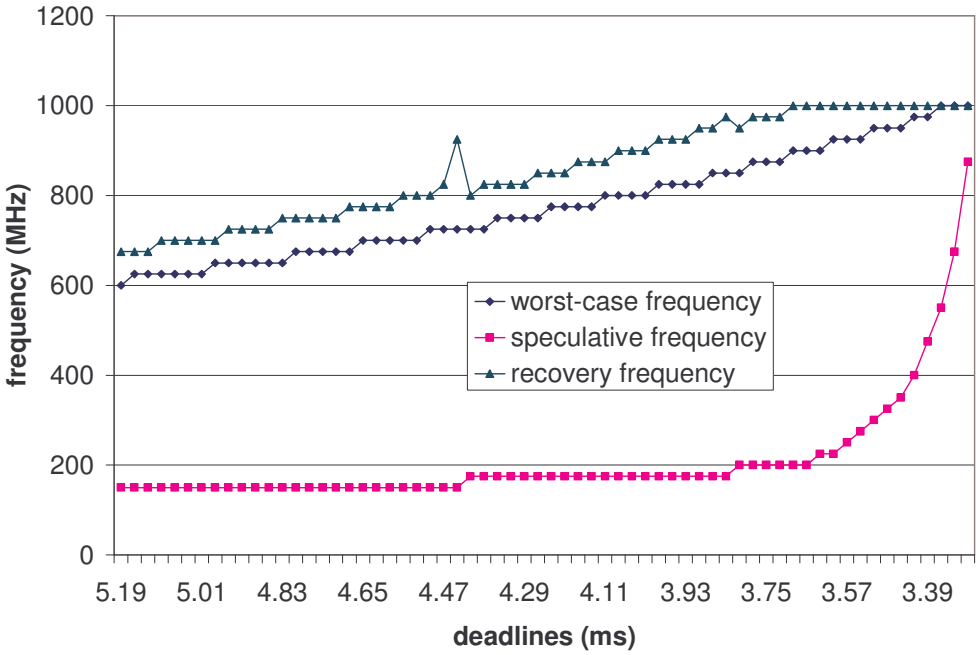


Figure 6-9. Worst-case, speculative, and recovery frequencies generated by the early-detection continuous-execution algorithm for decreasing deadlines for '*adpcm*'.

6.1.2 Varying PETs through on-line profiling

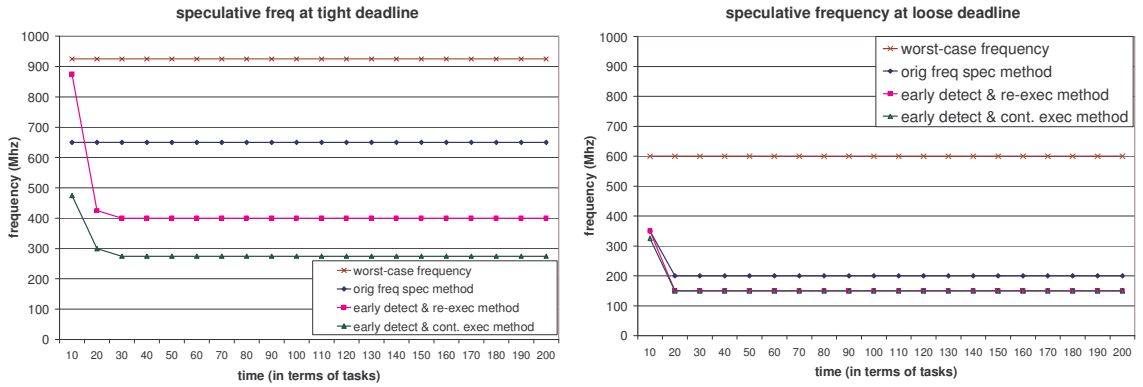
When a task is run for the first time, the predicted execution times are inflated because the actual execution times are unknown. In this set of experiments, the effect of the predicted execution times on the performance of the algorithms is studied. The benchmarks are assumed to be periodic tasks and are run 200 times consecutively. The predicted execution times are re-evaluated and the speculative and recovery frequencies are re-computed once every 10 times the task is executed.

Figure 6-10 (a) through Figure 6-15 (a) show the speculative frequencies, as the number of task instances increase over time, for the six benchmarks and a tight deadline. Initially, when the PETs are inflated, the original frequency speculation method performs

better than the early-detection re-execution method. But as the PETs get updated, both of the early-detection algorithms outperform the original frequency speculation method in all six benchmarks. The reason for the worse performance of the original frequency speculation method, relative to the other two methods, is that the mispredicted sub-task is executed entirely at the speculative frequency. This term turns out to be a large factor since the worst-case scenario is assumed while executing at the speculative frequency, which is low. In the other two algorithms, the penalty due to the mispredicted sub-task is less since the sub-task is either logically re-executed or finished at the recovery frequency, which is high. It is to be noted that the portion of the task that is executed at the speculative frequency is an irreducible component that is incurred in all three algorithms. Similarly, all the sub-tasks that follow the mispredicted sub-task are executed at the recovery frequency. It is the execution of the unfinished portion of the mispredicted task that is different in the three algorithms. In the original frequency speculation method, this portion is executed at the low speculative frequency, consuming a lot more time, whereas in the other algorithms, it is executed at the recovery frequency, consuming less time (hence producing slack that is utilized by lowering the speculative frequency).

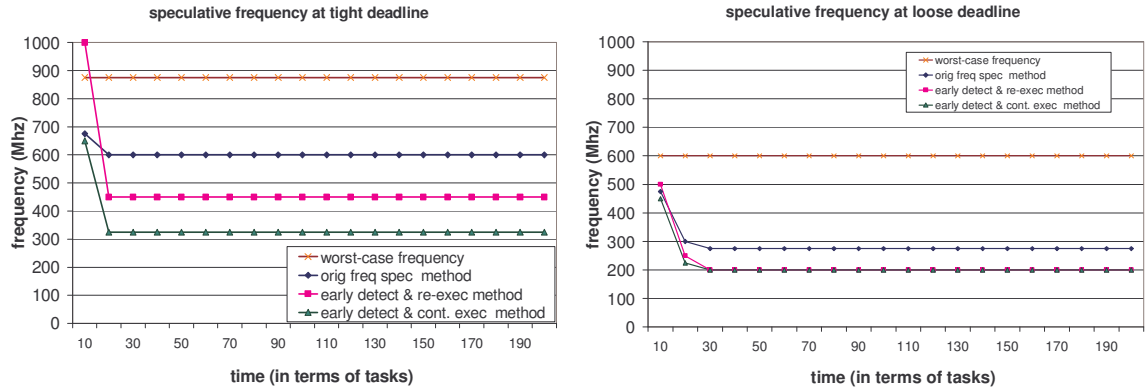
Note that the worst-case frequency stays constant even when the PETs vary. This is because the worst-case frequency is computed using only WCET numbers and is not affected by the PETs. It is seen that the worst-case frequency stays constant at 925Mhz for five benchmarks and at 875Mhz for one benchmark.

Figure 6-10 (b) through Figure 6-15 (b) show the speculative frequencies as the number of task instances increases over time, for six benchmarks and a loose deadline. It is seen that the frequency trends for loose deadlines are similar to trends for the tight deadlines. However, performance of the original frequency speculation algorithm after PETs have stabilized is closer to the other algorithms with loose deadlines than with tight deadlines. Also, the savings due to frequency speculation is less for a loose deadline than for a tight deadline. This can be explained by the fact that a loose deadline has more time and a lower worst-case frequency is sufficient.



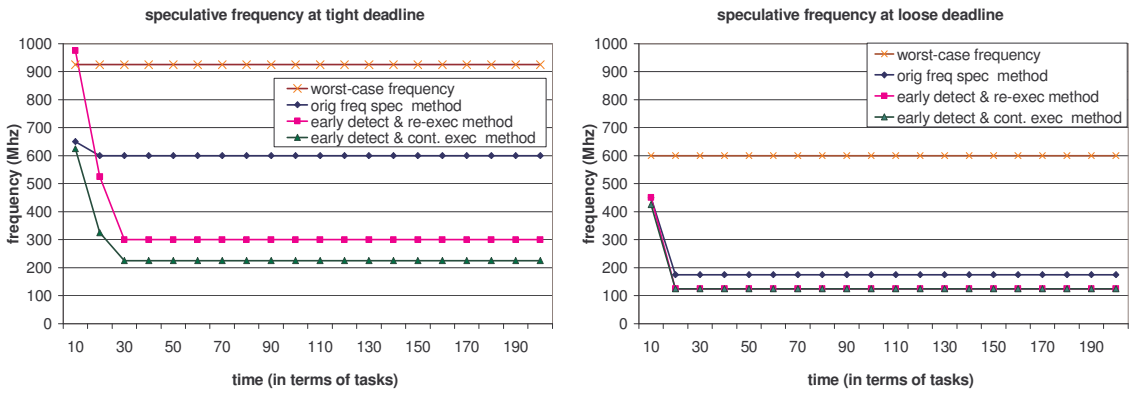
(a) tight deadline **(b) loose deadline**

Figure 6-10. Speculative frequency generated by different speculation algorithms for different PETs for 'adpcm', for (a) a tight deadline and (b) a loose deadline.



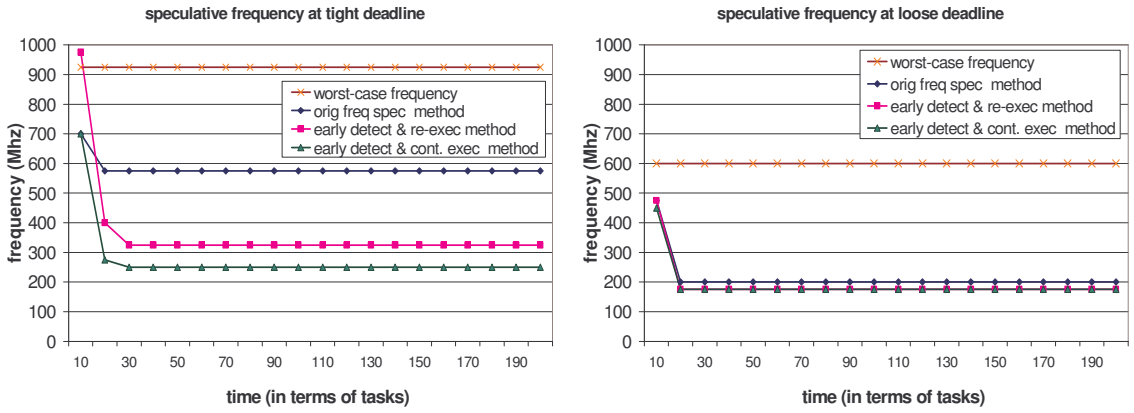
(a) tight deadline **(b) loose deadline**

Figure 6-11. Speculative frequency generated by different speculation algorithms for different PETs for 'cnt', for (a) a tight deadline and (b) a loose deadline.



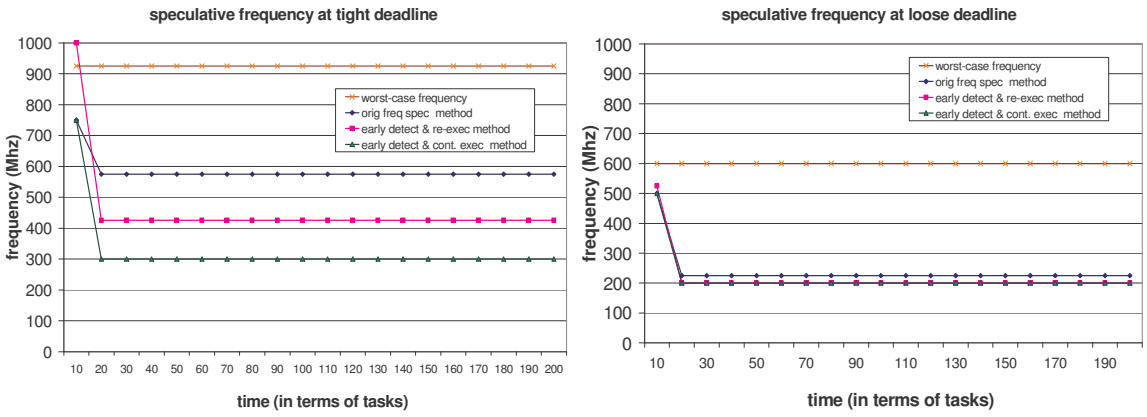
(a) tight deadline **(b) loose deadline**

Figure 6-12. Speculative frequency generated by different frequency speculation algorithms for different PETs for 'fft', for (a) a tight deadline and (b) a loose deadline.



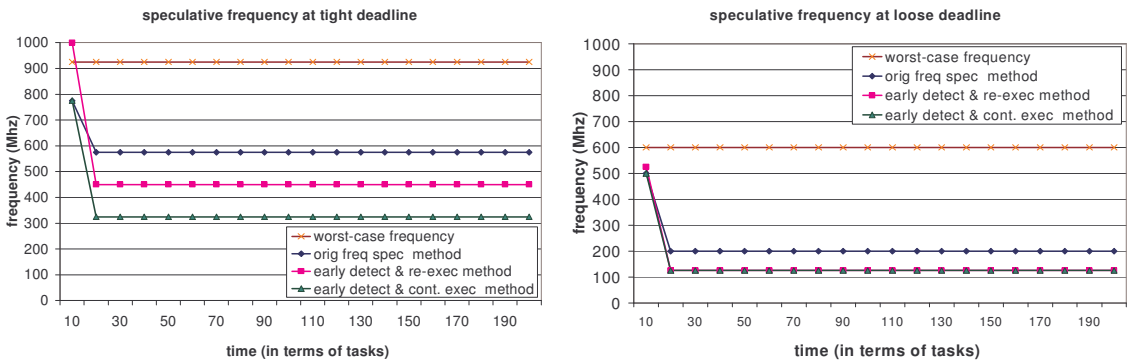
(a) tight deadline **(b) loose deadline**

Figure 6-13. Speculative frequency generated by different frequency speculation algorithms for different PETs for 'lms', for (a) a tight deadline and (b) a loose deadline.



(a) tight deadline **(b) loose deadline**

Figure 6-14. Speculative frequency generated by different frequency speculation algorithms for different PETs for 'mm', for (a) a tight deadline and (b) a loose deadline.



(a) tight deadline **(b) loose deadline**

Figure 6-15. Speculative frequency generated by different frequency speculation algorithms for different PETs for 'srt', for (a) a tight deadline and (b) a loose deadline.

6.1.3 Effect of WCET analysis

Worst-case analysis plays an important role in the computation of the speculative and recovery frequencies. Worst-case analysis also impacts the minimum deadlines that can be guaranteed. In this section, we investigate the effects of WCET on the performance of the various frequency speculation models.

WCET obtained from the static worst-case timing analysis tool is potentially pessimistic, since a simple pipeline was used for the analysis out of necessity. Therefore, the WCET is varied by subtracting successively larger amounts of time from it. The three algorithms derive the f_{spec} and f_{rec} for varying deadlines using this reduced WCET. The PETs are stable and are very close to actual execution times.

Figure 6-16 (a) through Figure 6-19 (a) show the speculative frequency for four different worst-case analysis scenarios, as a function of decreasing deadline, for the benchmark *cnt*. Similar trends are seen among all the other benchmarks (graphs have not

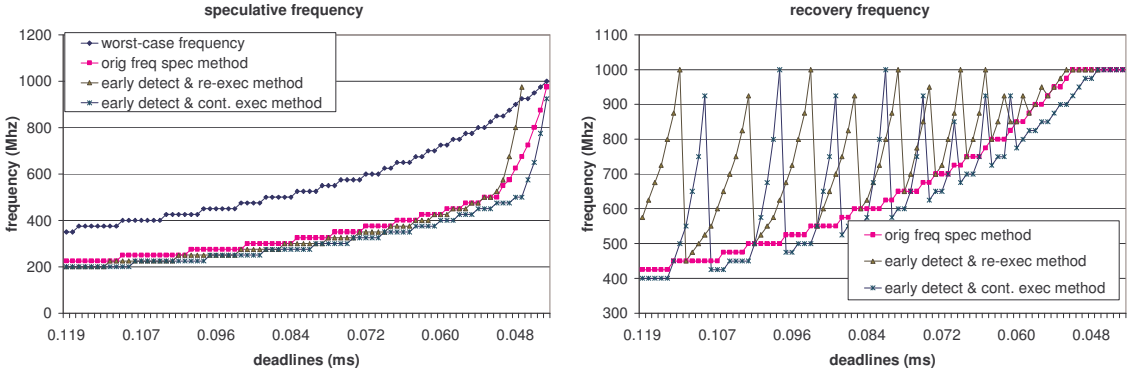
been included because of space considerations). Figure 6-16 (b) through Figure 6-19 (b) show the corresponding recovery frequencies.

The worst-case analysis is increasingly pessimistic, i.e., the first set of WCETs are the tightest (Figure 6-16) and the last set of WCET numbers are the loosest (Figure 6-19). WCETs from the static worst-case timing analysis tool available to us are considered to be the most pessimistic. These WCETs are used as base numbers. WCETs for tighter worst-case analysis are obtained by subtracting some time from the base numbers (how much is subtracted depends upon the size of the sub-tasks and the judgment of the author). Note that the WCETs derived this way are not provably safe. However, the intention of these experiments is to study the effect of tighter worst-case analysis on the performance of the frequency speculation algorithms and hence they are assumed to be safe for all practical purposes.

It is seen that the difference between the speculative and worst-case frequency increases as worst-case analysis becomes more pessimistic. For example, for benchmark *cnt* and a deadline of 0.1 ms, the difference between the speculative and worst-case frequency is 175Mhz for the tightest worst-case analysis. The difference is seen to be 250MHz, 325Mhz, and 375Mhz for progressively looser worst-case analysis for the same deadline. Similar trends are observed for all three algorithms. This can be explained by the fact that the worst-case frequency increases as the accuracy of the worst-case analysis decreases. Frequency speculation is more beneficial when the accuracy of the worst-case analysis is worse. However, note that fewer deadlines are guaranteed for pessimistic

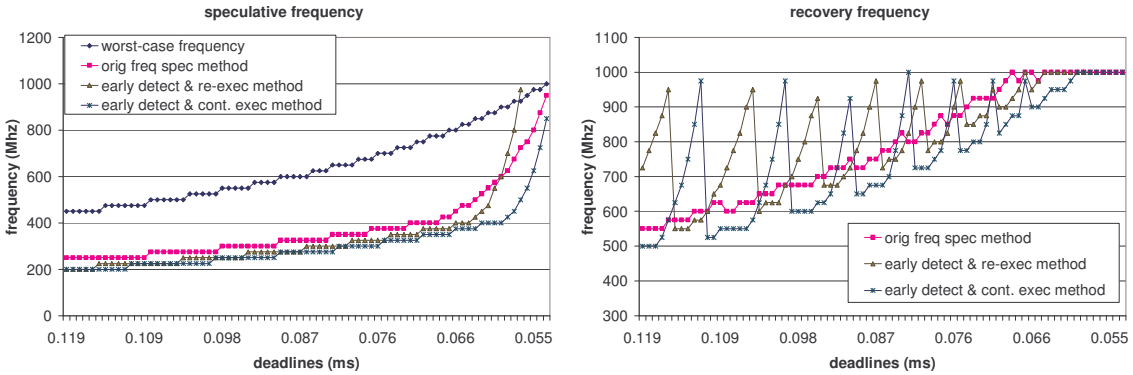
worst-case analysis. For example, for *cnt*, the tightest worst-case analysis guarantees a 0.043 ms deadline, whereas the loosest worst-case analysis cannot guarantee deadlines below 0.063, for the same task.

Recovery frequency is lowest for the tightest worst-case analysis and progressively increases with looser worst-case analysis. This is expected since higher WCET increases pressure on the recovery frequency, leading to an increased recovery frequency.



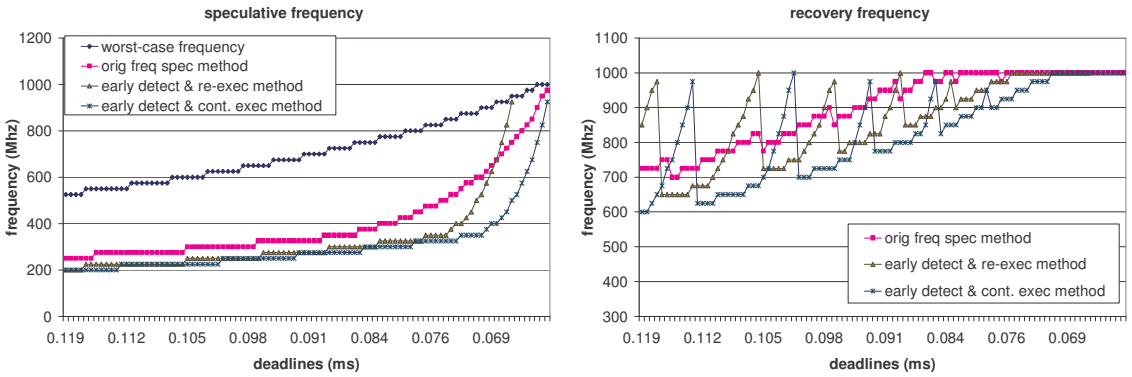
(a) **(b)**

Figure 6-16. (a) Speculative and (b) recovery frequencies generated by different algorithms assuming tight worst-case analysis, for *cnt*.



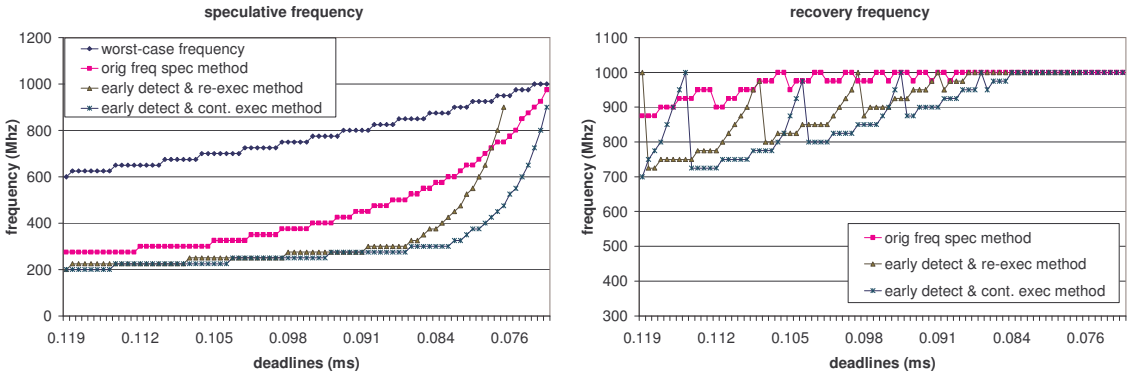
(a) **(b)**

Figure 6-17. (a) Speculative and (b) recovery frequencies generated by different algorithms assuming slightly pessimistic worst-case analysis, for *cnt*.



(a) (b)

Figure 6-18. (a) Speculative and (b) recovery frequencies generated by different algorithms assuming pessimistic worst-case analysis, for *cnt*.



(a) (b)

Figure 6-19. (a) Speculative and (b) recovery frequencies generated by different algorithms assuming highly pessimistic worst-case analysis, for *cnt*.

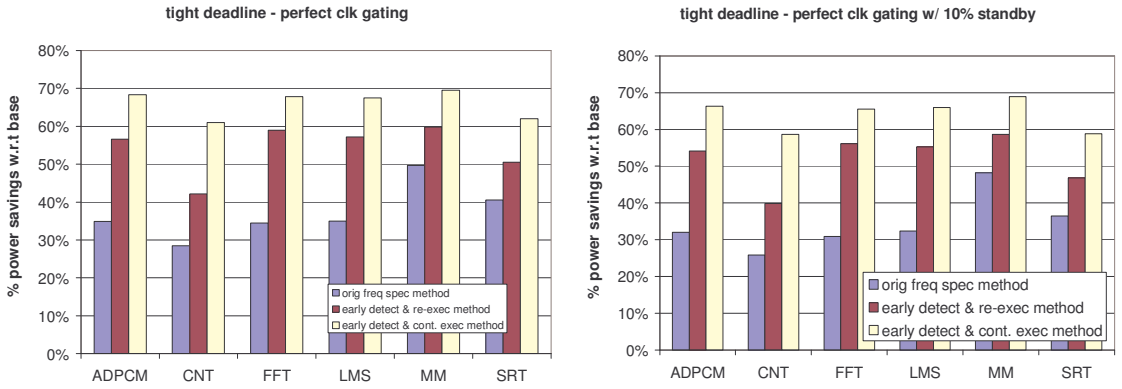
6.2 Power Results

The previous section presented savings in terms of frequency. With power models incorporated in the simulation environment, it is possible to show savings in terms of power and energy. This section presents savings in power and energy achieved by frequency speculation.

Figure 6-20 and Figure 6-21 show power and energy savings, respectively, for a tight deadline. Figure 6-22 and Figure 6-23 show power and energy savings, respectively, for a loose deadline. In each experiment, the benchmark is run 200 times consecutively as if it were a periodic task. Power/energy results are reported with respect to no speculation, i.e., the base case uses worst-case frequency. Results are reported assuming perfect clock gating and perfect clock gating with 10% standby power. Similarly, results have been reported for both tight and loose deadlines. Due to stable actual execution times, no sub-tasks mispredicted in any of the runs. However, we investigate the impact of misprediction on power/energy savings later.

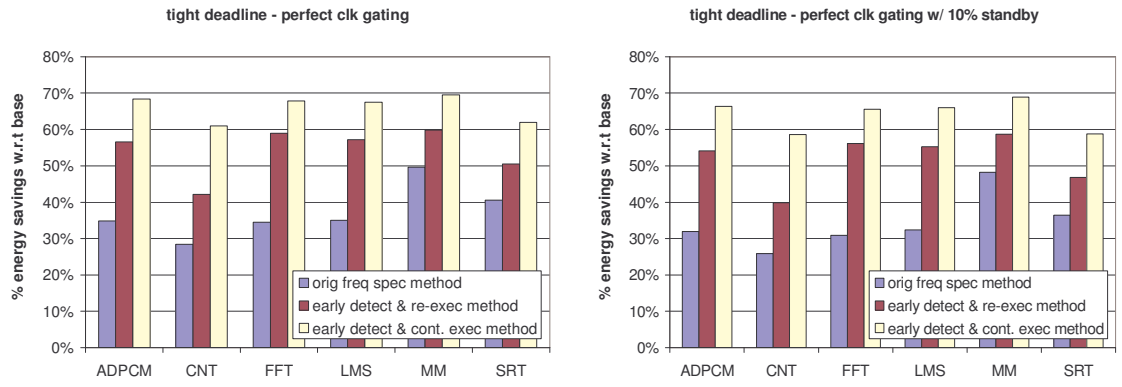
The first observation is that all the frequency speculation algorithms achieve significant power and energy savings, ranging from 25% to 68% for a tight deadline and 40% to 60% for a loose deadline. Comparing the efficiency of the three methods, we see that the original frequency speculation method achieves the least savings (30-50% for tight deadline, 45-58% for loose deadline) whereas the early-detection continuous-execution method achieves the most savings for both tight (60-70%) and loose deadlines (53-60%), as expected. However, the power savings of the three algorithms are closer for

a loose deadline. This is attributed to the fact that the speculative frequencies generated by the three algorithms are closer for loose deadlines.



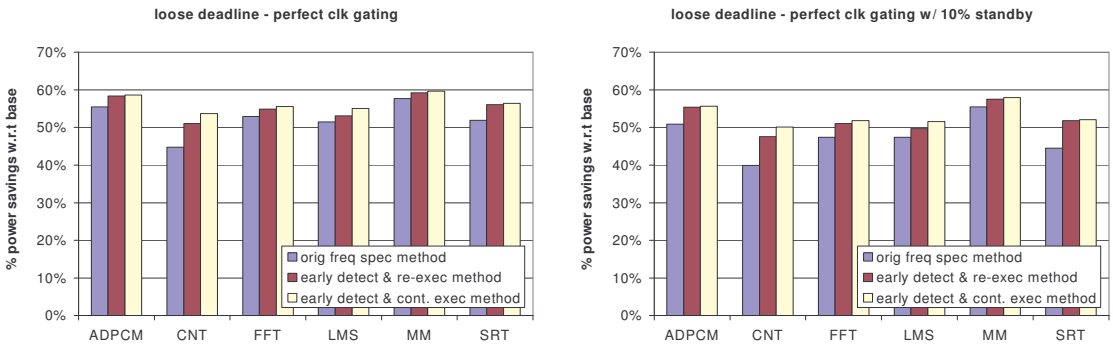
(a) Perfect clock gating **(b) Perfect clock gating with standby**

Figure 6-20. Power savings for different frequency speculation algorithms for a tight deadline, assuming (a) perfect clock gating and (b) perfect clock gating with standby power.



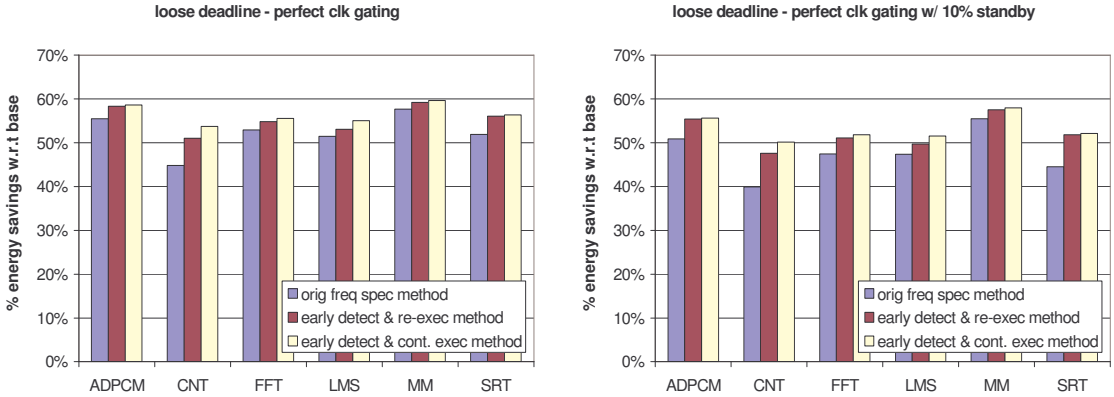
(a) Perfect clock gating **(b) Perfect clock gating with standby**

Figure 6-21. Energy savings for different frequency speculation algorithms for a tight deadline, assuming (a) perfect clock gating and (b) perfect clock gating with standby power.



(a) Perfect clock gating **(b) Perfect clock gating with standby**

Figure 6-22. Savings in power for different frequency speculation algorithms for a loose deadline assuming (a) perfect clock gating and (b) perfect clock gating with standby power.



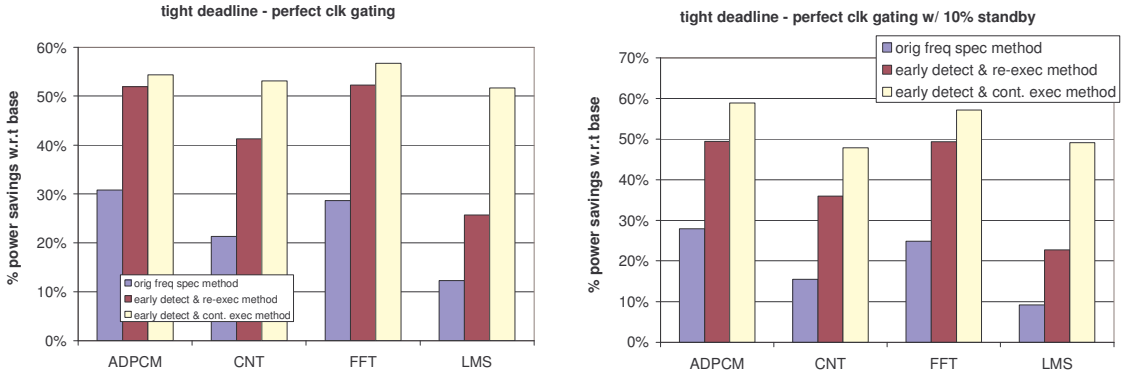
(a) Perfect clock gating **(b) Perfect clock gating with standby**

Figure 6-23. Savings in energy for different frequency speculation algorithms for a loose deadline assuming (a) perfect clock gating and (b) perfect clock gating with standby power.

Note that in the above experiments, there are no mispredictions and hence no transitions to the recovery frequency. We now show that frequency speculation can lower the power and energy consumption in spite of mispredictions. Since we are interested in microarchitectural uncertainty, we have used microarchitectural techniques such as flushing the instruction cache, data cache, and branch predictor to induce sub-task mispredictions. The task is consecutively executed 200 times and we flush the caches and branch predictor 20 times. This translates to roughly 20 tasks of the 200 tasks mispredicting (10% of the tasks transition to the recovery frequency). In this set of

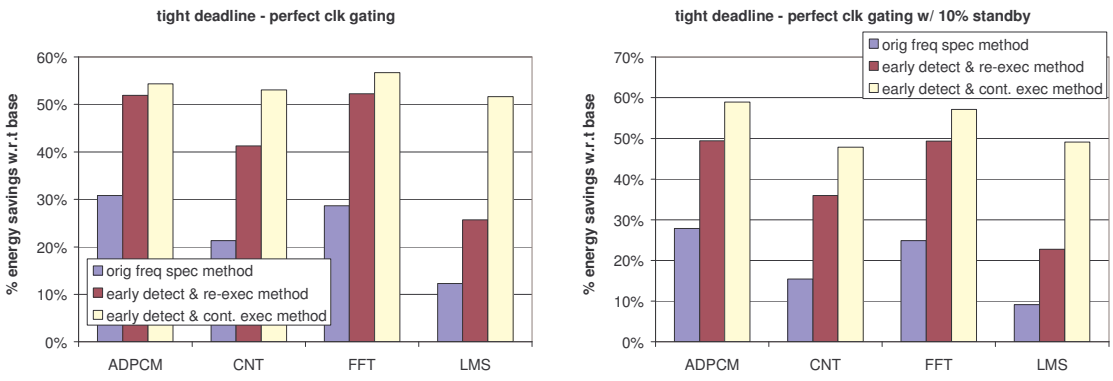
experiments, the last-N-based prediction heuristic has been used to set PETs unlike previous experiments (Figure 6-20 through Figure 6-23), which use histogram-based prediction.

Figure 6-24 shows power savings with respect to no frequency speculation. It can be seen that power savings goes down compared to the case in which there are no mispredictions. The savings goes down because approximately 10% of the tasks are partially executed at the higher recovery frequency, typically consuming more power than even the base case for those tasks. Similar trends are observed for energy savings (as shown in Figure 6-25).



(a) Perfect clock gating **(b) Perfect clock gating with standby**

Figure 6-24. Power savings for different frequency speculation algorithms for a tight deadline, assuming (a) perfect clock gating and (b) perfect clock gating with standby power, with 20 mispredictions out of 200 task executions.



(a) Perfect clock gating **(b) Perfect clock gating with standby**

Figure 6-25. Energy savings for different frequency speculation algorithms for a tight deadline, assuming (a) perfect clock gating and (b) perfect clock gating with standby power, with 20 mispredictions out of 200 task executions.

6.3 Effects of Sub-task Selection

The effect of sub-task selection on frequency speculation is investigated in this section. Sub-tasks can be balanced or unbalanced in terms of the execution times. In this section, we experiment with less balanced sub-tasks. This is achieved by concatenating equal-sized sub-tasks to form a larger sub-task, and leaving other sub-tasks as is. Unbalanced tasks can be increasingly, decreasingly, or randomly unbalanced. In an increasingly unbalanced task, the sizes of the sub-tasks in the task progressively increase. The sizes of sub-tasks progressively decrease in the case of a decreasingly unbalanced task. The sub-task sizes do not exhibit any particular trend in the case of a randomly unbalanced task. Only three of the six benchmarks have been used for these experiments.

Figure 6-26 through Figure 6-28 show the trends for the speculative and recovery frequencies generated by the three algorithms for different sub-task selections, for the benchmark *adpcm*. Three types of sub-task selection are represented: balanced,

increasingly unbalanced, and decreasingly unbalanced. That is, the sizes of the sub-tasks are similar, progressively increasing, or progressively decreasing, respectively.

It is interesting to compare the behavior of the frequencies for different sub-task selection methods. For all three frequency speculation algorithms, we can see that decreasingly unbalanced sub-task selection is least efficient. It always yields a higher speculative frequency than balanced and increasingly unbalanced sub-task selection. For loose deadlines, the efficiency of balanced sub-tasks is very close to (and often better than) that of increasingly unbalanced sub-task selection. However, as the deadline is tightened, the increasingly unbalanced sub-task selection yields the lowest speculative frequency. This can be explained as follows. In the scenario of the first sub-task mispredicting, all subsequent sub-tasks are executed assuming the worst-case number of cycles. This means that more work has to be done in this case compared to the scenarios of the other sub-tasks mispredicting. Hence, the scenario of the first sub-task mispredicting dictates the initial speculative and recovery frequencies. The ratio of the misprediction penalty to the overall task size is lowest in the increasingly balanced case in the case of the first sub-task mispredicting. Misprediction penalty is the extra time/cycles due to a misprediction. In the increasingly balanced method, since the size of the first sub-task is small, the misprediction penalty is small too. Hence increasingly unbalanced yields the lowest speculative frequency. On the other hand, balanced sub-tasks selection yields a larger first sub-task (hence a larger misprediction penalty) and a higher speculative frequency results. Decreasingly unbalanced has an even larger first

sub-task and higher speculative frequency. Again, this behavior can again be attributed to the ratio of the misprediction penalty to the overall task size.

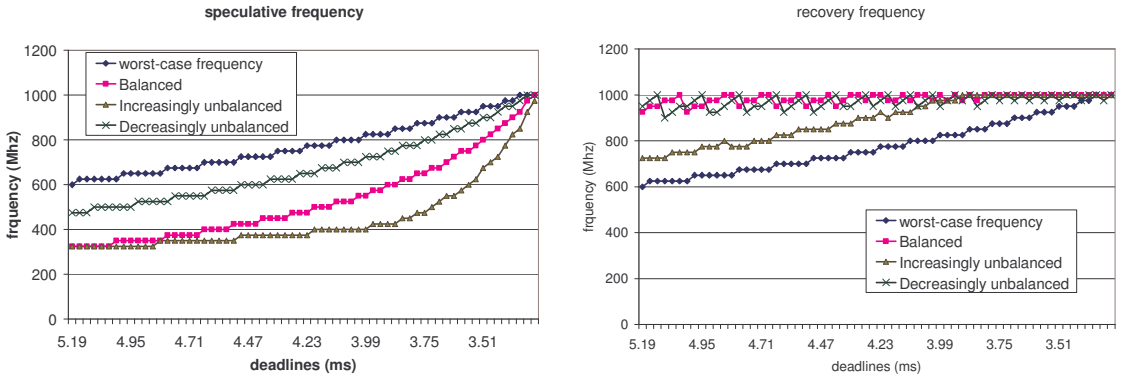


Figure 6-26. (a) Speculative and (b) recovery frequencies generated by the original frequency speculation algorithm for various sub-task selection methods, for 'adpcm '.

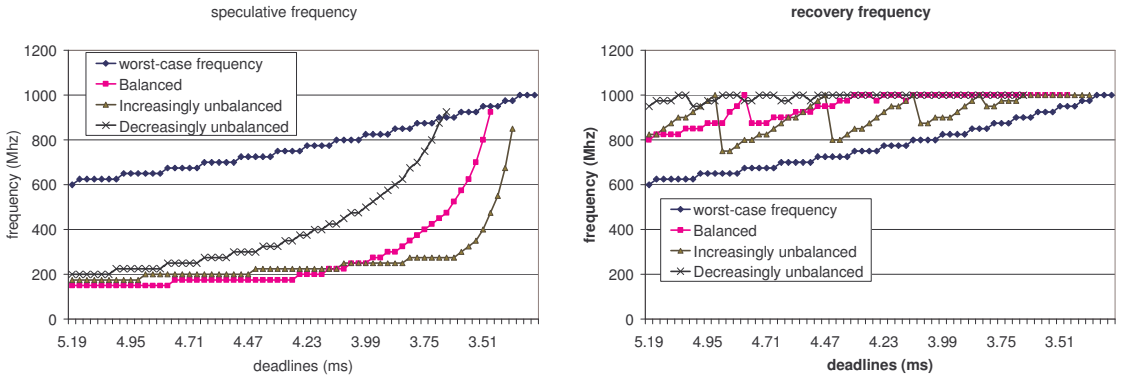


Figure 6-27. (a) Speculative and (b) recovery frequencies generated by the early-detection re-execution algorithm for various sub-task selection methods, for 'adpcm '.

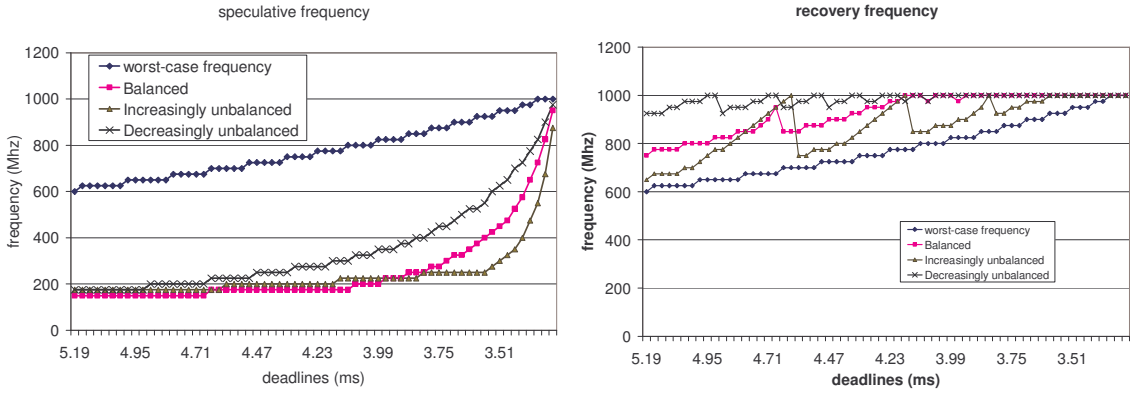


Figure 6-28. (a) Speculative and (b) recovery frequencies generated by early-detection continuous-execution algorithm for various sub-task selection methods, for 'adpcm '.

Chapter 7 Summary and Future Work

In hard real-time systems, system frequency is designed on the basis of worst-case execution time to guarantee safe operation. Worst-case analysis of a high-performance microarchitecture is pessimistic leading to high clock frequencies, higher than actually needed in practice.

Frequency speculation is a method for safely reducing the frequency on the basis of typical execution time. The real-time task is divided into sub-tasks. Their predicted execution times and worst-case execution times are used to compute a speculative frequency and a safe recovery frequency. The task is attempted at the speculative frequency and the sub-tasks are expected to complete within their predicted execution times. If a sub-task does not complete within its predicted execution time, the system takes corrective measures. The remaining sub-tasks are run at the recovery frequency to finish the task before the deadline, in spite of the interim misprediction.

This thesis proposes two new frequency speculation algorithms that improvise on the originally proposed frequency speculation algorithm. A drawback of the original frequency speculation algorithm is that a sub-task misprediction is detected only after completing the sub-task. This thesis showed that a misprediction could be detected earlier through the use of a watchdog timer. Since recovery can be initiated earlier, in the middle of the mispredicted sub-task, early detection introduces extra slack that can be used to lower the speculative frequency even further.

However, early detection poses the new problem of bounding the amount of time needed to complete the work in the mispredicted sub-task. The two new algorithms differ in how the unfinished work is bounded. The first algorithm, called early-detection logical re-execution algorithm, conservatively bounds the execution time of the unfinished portion using the worst-case execution time of the entire sub-task. The second, called early-detection continuous-execution, uses more sophisticated analysis to derive a much tighter bound. Results confirm that both early-detection algorithms outperform the late-detection algorithm.

We have also implemented a detailed simulation framework to study frequency speculation. We have demonstrated the feasibility of sub-task selection for real benchmarks. On-line profiling ensures that the predicted execution times are continuously updated and the speculative and recovery frequencies are as low as possible. The software and hardware support that is needed for frequency speculation has been identified and described in detail.

From the research in this thesis, we are able to draw the following conclusions.

- Static frequency speculation yields up to 70% savings in power and energy.
- The early-detection continuous-execution algorithm has the highest power savings (60-70%), followed by the early-detection logical re-execution algorithm (42-60%). The original frequency speculation algorithm, which uses late detection, has the least savings (30-58%). These results demonstrate that early detection is advantageous over late detection, confirming our original hypothesis.

- The three algorithms perform similarly for loose deadlines. This is because the ratio of the misprediction penalty (the only differentiating factor among the three algorithms) to the overall deadline is small at loose deadlines.
- Power savings increases as the pessimism of worst-case analysis increases. This is particularly significant in the context of future trends: WCET pessimism is expected to increase as microarchitectural complexity increases.
- Sub-task selection affects the power savings achieved by static frequency speculation. Increasingly unbalanced sub-tasks have the most potential for power/energy savings, followed by balanced sub-tasks, followed by decreasingly unbalanced sub-tasks.

Static worst-case timing analysis of a complex microarchitecture, such as the one used in this thesis, is potentially intractable. A tool that safely and accurately analyzes a complex microarchitecture is not known to exist at this time. We are investigating a new method that eliminates the need to do explicit worst-case timing analysis of complex architectures.

Future work also includes the following.

- It has been observed that increasing the statically generated speculative frequency to the next available frequency can significantly decrease the recovery frequency under certain conditions. We will study the trade-offs between the speculative and recovery frequencies for such cases.

- In this thesis, a single recovery frequency was used irrespective of which sub-task mispredicts, yet recovery frequency can be optimized if the mispredicted sub-task were known in advance. We propose statically generating a table of recovery frequencies, and selecting one from the table at run-time depending on which sub-task mispredicts.
- It might be possible to anticipate mispredictions in future sub-tasks based on the direction of key branches (depending on how control flow resolves). We will develop mechanisms to anticipate the behavior of later sub-tasks. This information can be used to take actions earlier and thereby optimize the recovery frequency.
- We will explore frequency speculation approaches that combine various aspects of static and dynamic schemes. Particularly, we are interested in employing dynamic schemes to reduce the recovery frequency after a misprediction has occurred. For example, the system could resume speculation after a misprediction if timeliness is regained while executing at the recovery frequency. Alternatively, slack generated while executing at the recovery frequency could be used to safely lower the recovery frequency dynamically.
- We will investigate the impact of the number of sub-tasks on frequency speculation.
- We will explore the feasibility of sub-task selection for more complex benchmarks. We will also explore automation of sub-task selection.
- We will extend static frequency speculation to work in the context of multitasking and pre-emption.

Bibliography

[1] N. AbouGhazaleh, D. Mossé, B. Childers, and R. Melhem. Toward The Placement of Power Management Points in Real Time Applications, *COLP'01 (Workshop on Compilers and Operating Systems for Low Power), 2001*

[2] R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. *15th Real-Time Systems Symposium*, 1994.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. *27th Int'l Symposium on Computer Architecture*, June 2000.

[4] D. Burger, T. Austin, and S. Bennett. Evaluating Future Microprocessors: The Simplescalar Toolset. Tech. Rep. CS-TR-96-1308, CS Dept., Univ. of Wisconsin - Madison, July 1996.

[5] M. Harmon, T. P. Baker, and D. B. Whalley. A retargetable technique for predicting execution time. *Real-Time Systems Symposium*, pp. 68–77, Dec. 1992.

[6] C. A. Healy, R. D. Arnold, F. Mueller, D. Whalley, and M. G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, Jan. 1999.

[7] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. *Real-Time Systems Symposium*, pp. 288–297, Dec. 1995.

[8] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. *1st Int'l Conference on Mobile Computing and Networking*, Nov. 1995.

[9] D. Grunwald, P. Levis, C. Morrey III, M. Neufeld, and K. Farkas. Policies for dynamic clock scheduling. *Symp. on Operating Systems Design and Implementation*, Oct. 2000.

[10] I. Hong, M. Potkonjak, and M. Srivastava. On-line scheduling of hard real-time tasks on variable voltage processor. *Int'lConference on Computer-Aided Design*, Nov. 1998.

[11] I. Hong, G. Qu, M. Potkonjak, and M. Srivastava. Synthesis techniques for low-power hard real-time systems on variable voltage processors. *19th Real-Time Systems Symposium*, Dec. 1998.

[12] C. Krishna and Y. Lee. Voltage clock scaling adaptive scheduling techniques for low power in hard real-time systems. *6th Real-Time Technology and Applications Symposium*, May 2000.

[13] Y. Lee and C. Krishna. Voltage clock scaling for low energy consumption in real-time embedded systems. *6th Int'l Conf. on Real-Time Computing Systems and Applications*, Dec. 1999.

[14] J. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with PACE. *Proceedings of the ACM SIGMETRICS 2001 Conference*, June 2001.

[15] D. Mossé, H. Aydin, B. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. *Workshop on Compilers and Operating Systems for Low Power*, Oct. 2000.

[16] F. Mueller. Static Cache Simulation and its Applications. Ph.D. Thesis, Dept. of CS, Florida State Univ., July 1994.

[17] F. Mueller. Generalizing timing predictions to set-associative caches. *EuroMicro Workshop on Real-Time Systems*, pp. 64–71, June 1997.

[18] F. Mueller. Timing predictions for multi-level caches. ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems, pp. 29–36, June 1997.

[19] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):209–239, May 2000.

[20] F. Mueller, D. B. Whalley, and M. Harmon. Predicting instruction cache behavior. *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.

[21] T. Pering, T. Burd, and R. Brodersen. The simulation of dynamic voltage scaling algorithms. *Symp. on Low Power Electronics*, 1995.

[22] E. Rotenberg. Using Variable-MHz Microprocessors to Efficiently Handle Uncertainty in Real-Time Systems. *34th International Symposium on Microarchitecture,* December 2001.

[23] Y. Shin, K. Choi, and T. Sakurai. Power optimization of real-time embedded systems on variable speed processors. *Int'l Conf. on Computer-Aided Design*, 2000.

[24] R. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and set-associative caches. *Real-Time Technology and Applications Symposium*, pp. 192–202, June 1997.

[25] R. T. White, F. Mueller, C. Healy, D. Whalley, and M. G. Harmon. Timing analysis for data and wrap-around fill caches. *Real-Time Systems*, 17(2/3):209–233, Nov. 1999.

[26] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. *1st Symp. on Operating Systems Design and Implementation*, Nov. 1994.

[27] http://developer.intel.com/design/intelxscale/benchmarks.htm

[28] http://www.intel.com/design/intelxscale/

[29] "C-lab: WCET Benchmarks," http://www.c-lab.de/home/en/download.html.

A-1 Appendix

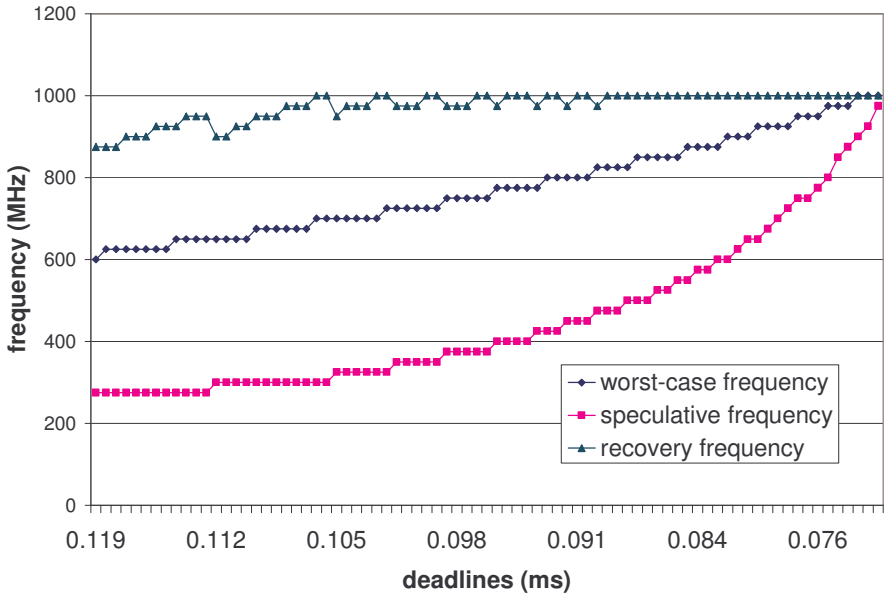


Figure A-1. Worst-case frequency, speculative, and recovery frequencies generated by the original frequency speculation algorithm for varying deadlines, for *'cnt'*.

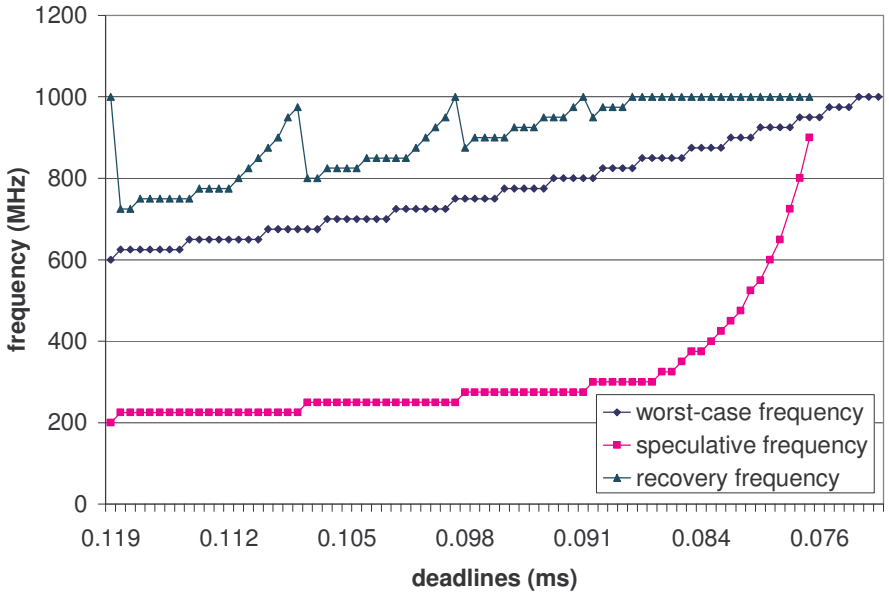


Figure A-2. Worst-case frequency, speculative, and recovery frequencies generated by the early-detection re-execution algorithm for varying deadlines, for *'cnt'*.

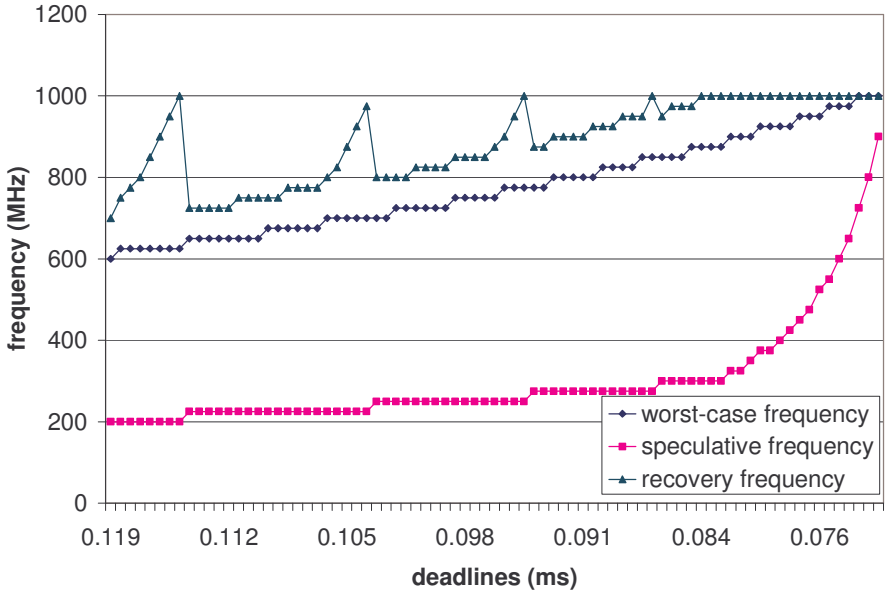


Figure A-3. Worst-case frequency, speculative, and recovery frequencies generated by the early-detection continuous-execution algorithm for decreasing deadlines, for *'cnt'*.

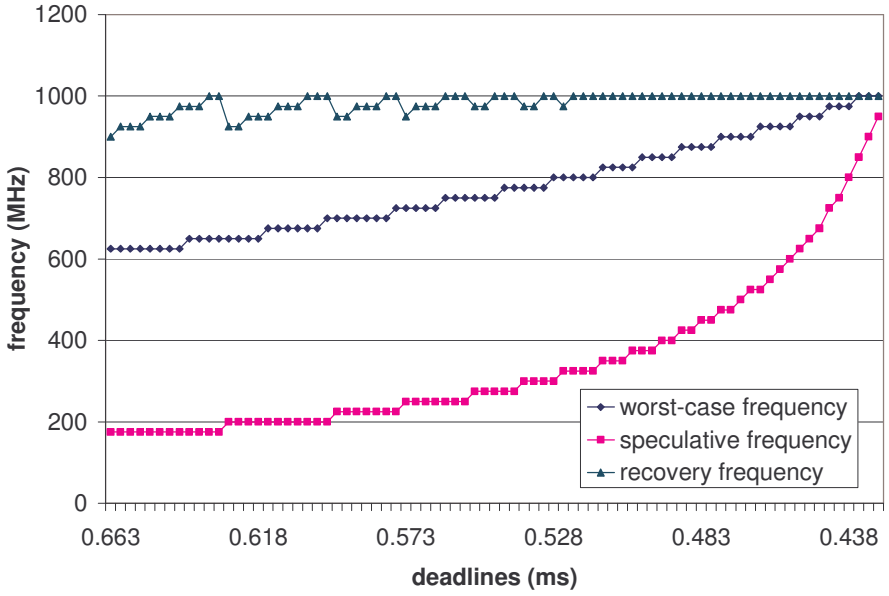


Figure A-4. Worst-case frequency, speculative, and recovery frequencies generated by the original frequency speculation algorithm for varying deadlines, for *'fft'*.

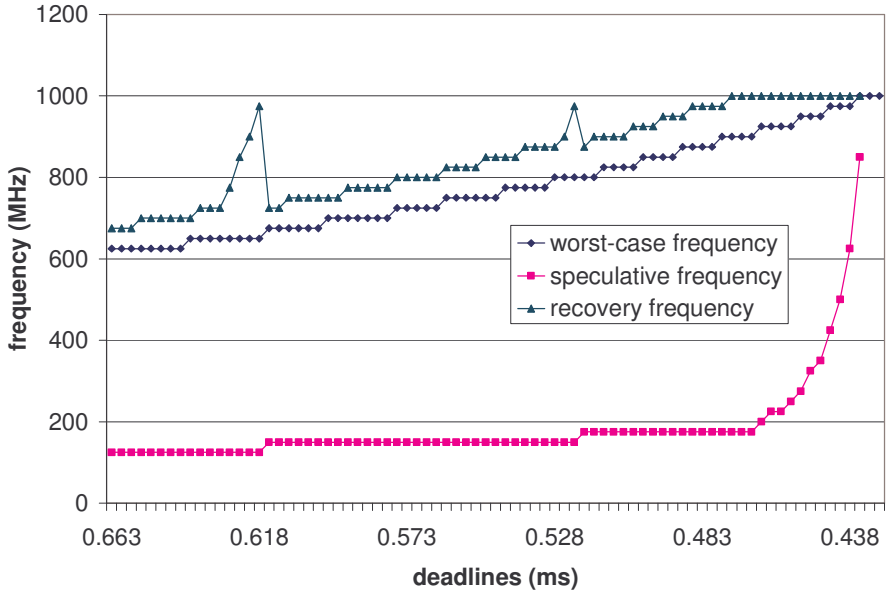


Figure A-5. Worst-case frequency, speculative, and recovery frequencies generated by the early-detection re-execution algorithm for varying deadlines, for *'fft'*.

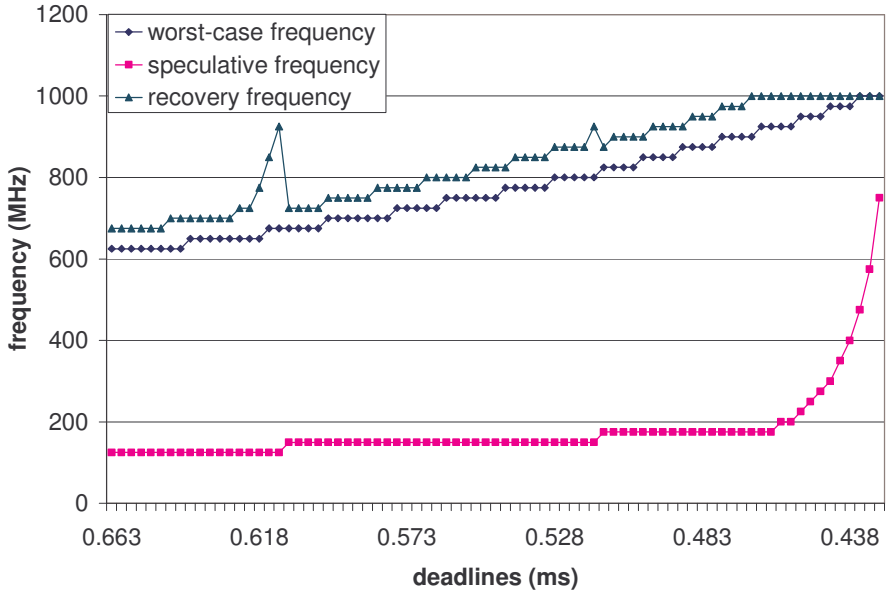


Figure A-6. Worst-case frequency, speculative, and recovery frequencies generated by the early-detection continuous-execution algorithm for varying deadlines, for *'fft'*.

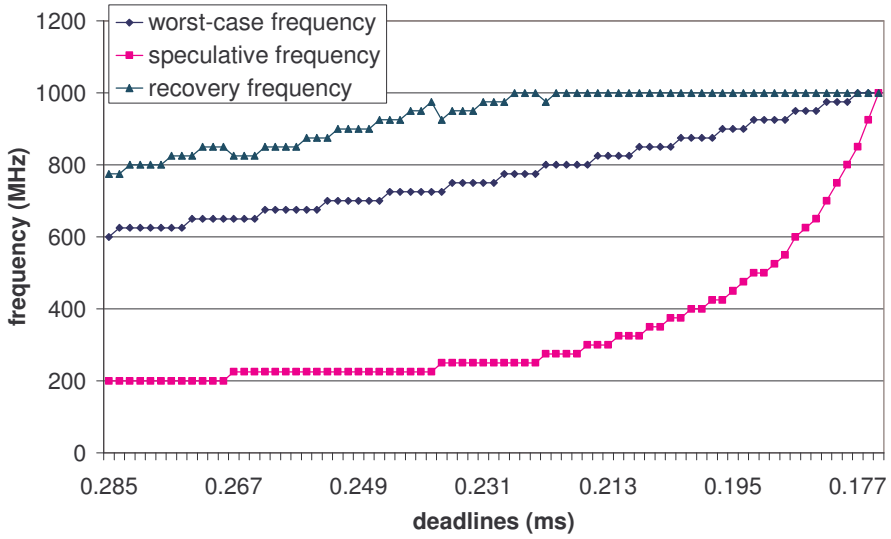


Figure A-7. Worst-case frequency, speculative, and recovery frequencies generated by the original frequency speculation algorithm for varying deadlines, for *'lms'*.

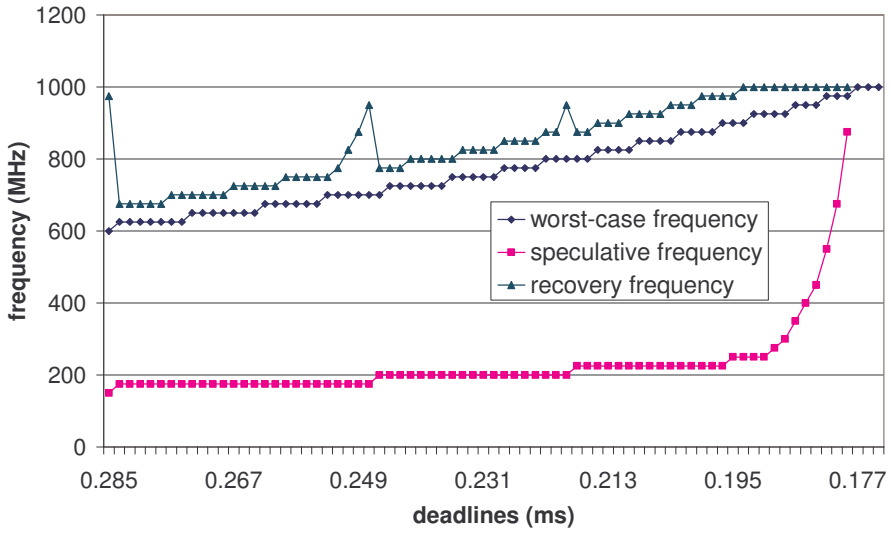


Figure A-8. Worst-case frequency, speculative, and recovery frequencies generated by the early-detection re-execution algorithm for varying deadlines, for *'lms'*.

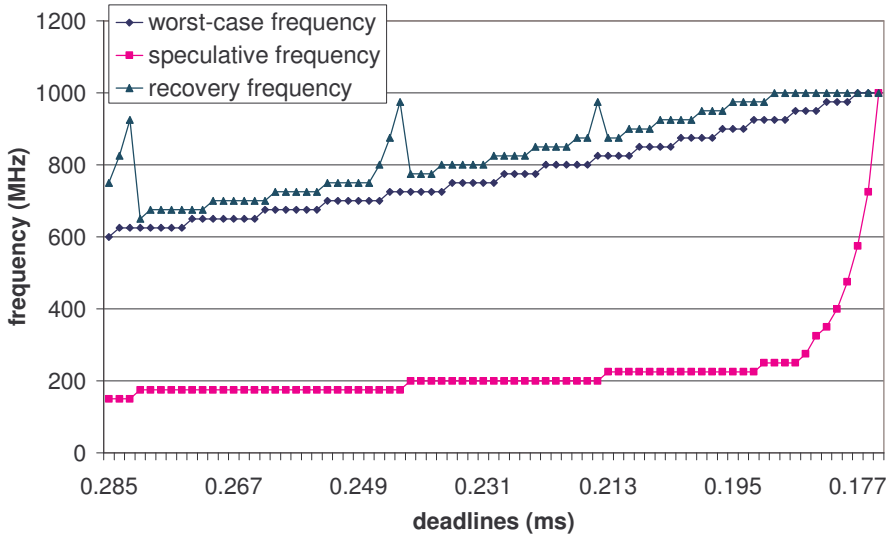


Figure A-9. Worst-case frequency, speculative and recovery frequencies generated by the early-detection continuous-execution algorithm for varyingdeadlines, for *'lms'*.

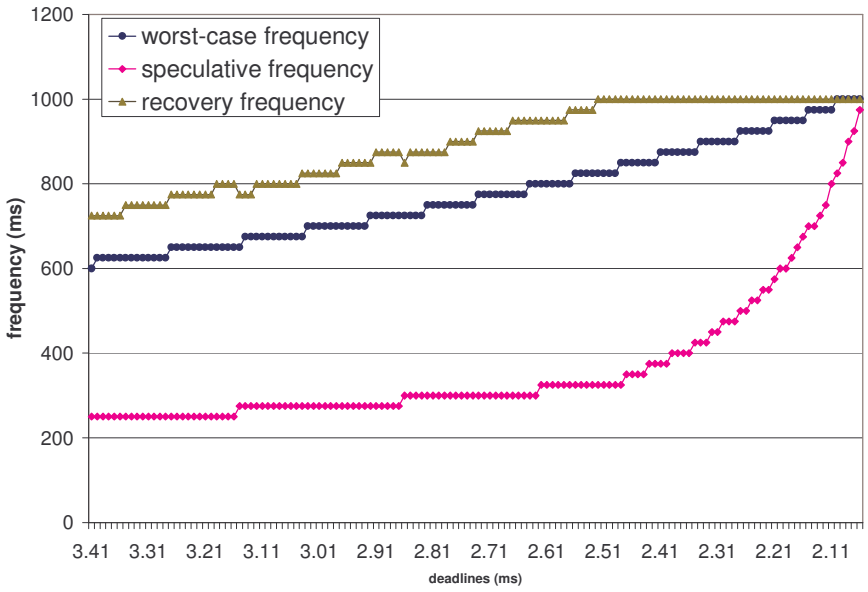


Figure A-10. Worst-case frequency, speculative, and recovery frequencies generated by the original frequency speculation algorithm for varying deadlines, for *'mm'*.

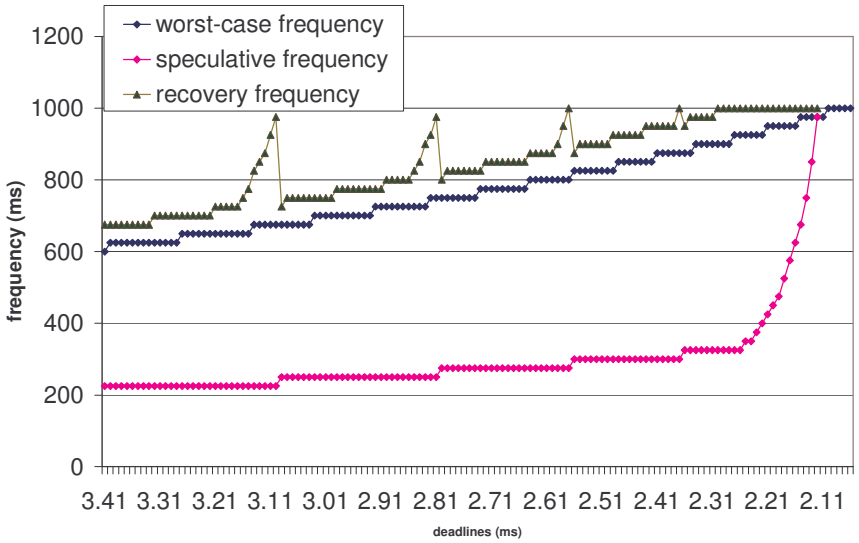


Figure A-11. Worst-case frequency, speculative, and recovery frequencies generated by the early-detection re-execution algorithm for varying deadlines, for *'mm'*.

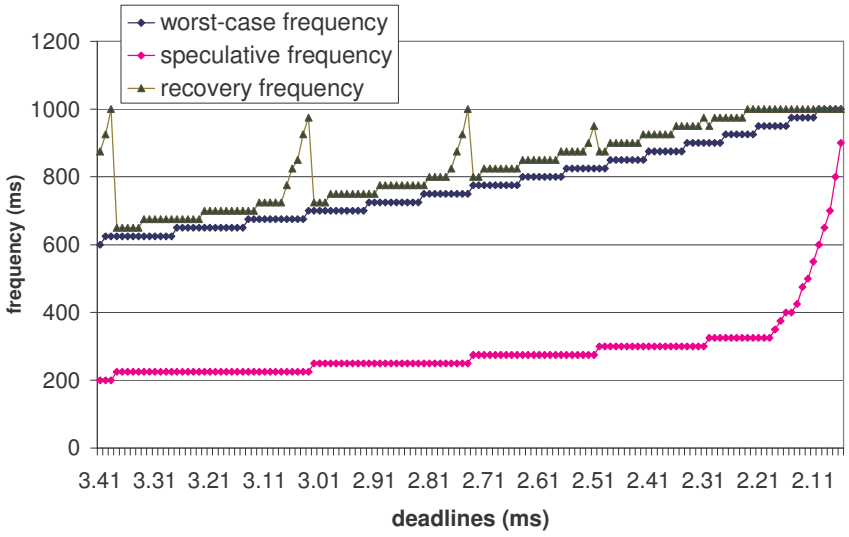


Figure A-12. Worst-case frequency, speculative, and recovery frequencies generated by the early-detection continuous algorithm for varying deadlines, for *'mm'*.

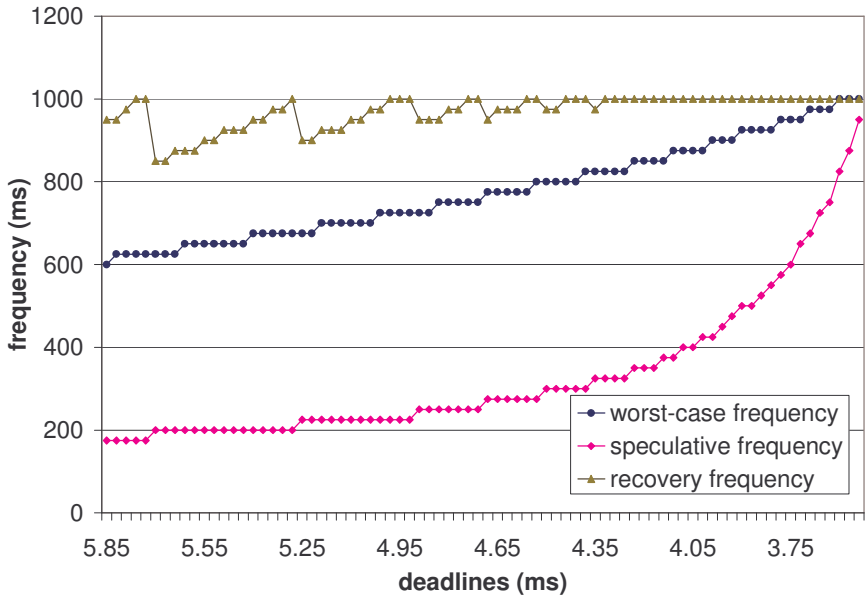


Figure A-13. Worst-case frequency, speculative, and recovery frequencies generated by the original frequency speculation algorithm for varying deadlines, for *'srt'*.

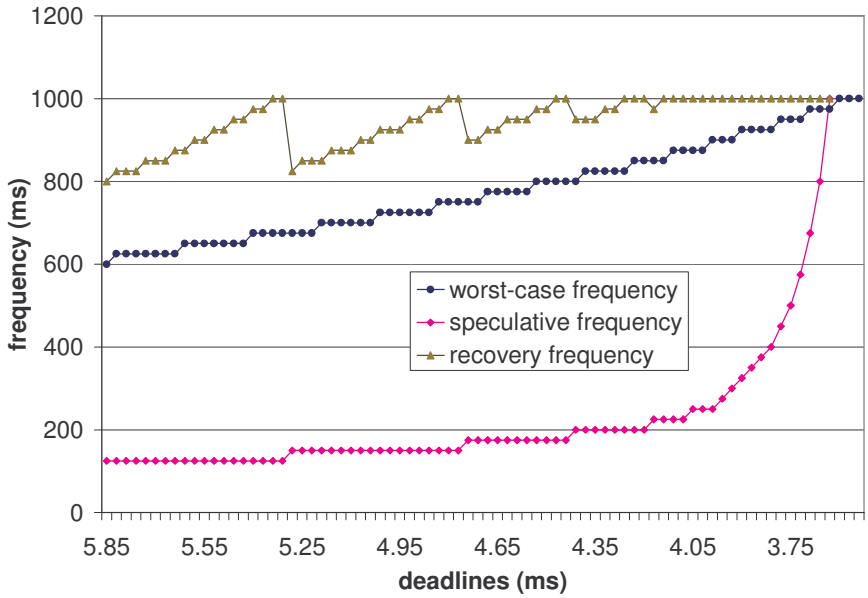


Figure A-14. Worst-case frequency, speculative, and recovery frequencies generated by the early detection re-execution algorithm for varying deadlines, for *'srt'*.

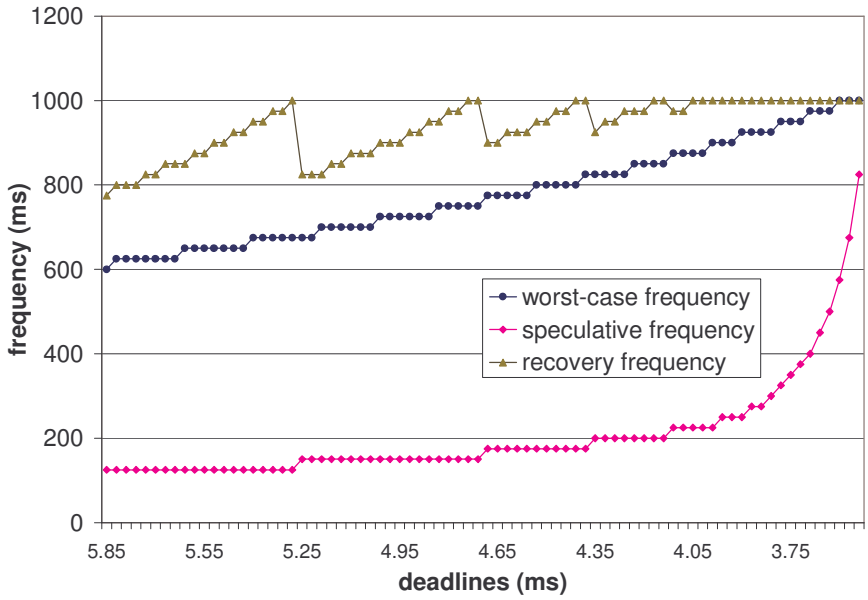


Figure A-15. Worst-case frequency, speculative, and recovery frequencies generated by the early detection continuous algorithm for varying deadlines for *'srt'*.