

## Chapter 1

# Introduction

Measuring performance is important for tuning and understanding program behaviour. This thesis concentrates on the development of performance monitoring hardware in cache-coherent multiprocessors, what it should measure, and its uses. Additionally, a flexible implementation of the performance monitor for NUMAchine [Vranesic95] is described.

## 1.1 Motivation

Microprocessors have improved in speed at a tremendous rate in the last decade. The improvement has come from the combination of advances in silicon processing technology and architectural progress. However, the performance gap between main memory and microprocessors has widened, and it is apparent that this trend will continue. This speed discrepancy necessitates usage of cache memory. Cache memory is effective because programs often exhibit locality of reference. Yet some programs do not exhibit locality, and others that do have locality can fail to have requisite data in cache due to caching policies. Such programs exhibit significant performance loss, and hence cache behaviour is a prime candidate for performance tuning.

The problem of poor cache performance is worsened in shared-memory multiprocessors. Memory access times in multiprocessors are typically much longer than in uniprocessors due to longer paths through the communication network, additional arbitration logic, and network or memory contention. Cache effectiveness is further decreased with cache coherence, because a processor that modifies data must first remove any shared copies from the caches of other processors. Despite these performance problems, cache-coherent multiprocessors are attractive parallel systems because they provide a simple programming model.

Poor cache behaviour makes parallel applications difficult to scale and speedups are poor — even when other sources of performance loss, such as load imbalance, are minimized. To improve performance the programmer needs insight into the program’s behaviour and the sources of performance loss. For example, knowing the number of references that miss in the cache gives the programmer a specific target to minimize. A more detailed breakdown, such as knowing which code segments suffered most of the misses, would be even more helpful.

To assist with performance tuning, a variety of previously developed tools can be used. Software such as gprof [Graham82] measures execution time spent in each procedure of a program being monitored, but its results can be misleading since it is not clear whether a procedure is simply inefficient or if caching problems occur. Another tool, Mtool [Goldberg93], provides more detail by breaking up procedure execution time into compute time, memory overhead, and synchronization overhead, but it is still at the procedure level. Both gprof and Mtool are intrusive and can introduce a significant *probe effect*. Other tools, such as CProf [Lebeck94] and MemSpy [Martonosi95], go into greater detail by performing cache-level simulations to identify exactly where cache misses occur, but they exhibit enormous runtime overhead; MemSpy slows parallel execution by a factor of about 200 [Martonosi95]. These tools are helpful, but they can be too abstract, intrusive, or very slow.

In contrast to performance tuning software, dedicated hardware can be used to alleviate most of the problems. Performance monitoring hardware can be made *nonintrusive* and, consequently, allow applications to run at full speed. Also, collected data can be as fine-grained as required to assist tuning. Additionally, events that are unobservable or inconvenient to measure in software, such as peak memory contention or response time to invoke an interrupt handler, can be done effectively in hardware. Consequently, hardware can expand the variety and detail of measurements available with no impact on application performance, providing a ‘best of both worlds’ option that is not available with software-based approaches.

Although motivated by tuning, a hardware performance monitor has additional uses. For example, it can help programs adapt to their run-time situation by dynamically choosing whether prefetching will improve performance [Horowitz95]. Also, it can assist computer architects by collecting data normally obtained from slow system simulations. Similarly, collected data can be used to help verify mathematically-abstract system models, characterize workloads for the models, and even generate benchmark suites. For example, one supercomputer site is creating such a suite based on one year of data from a Cray Y-MP [Gao95].

## 1.2 Objectives

Although a hardware performance monitor adds to the cost of designing and constructing a multiprocessor, its use can add significant value. In this thesis, factors which influence multiprocessor performance are identified and, where possible, hardware features to measure these factors are proposed. The primary objectives of the hardware features are to identify the causes of software performance loss and aid multiprocessor research while maintaining low cost and remaining nonintrusive. There are many measurements which can be made, so prudence is required. When selecting what features are important to measure in hardware, the primary objectives are carefully considered. To demonstrate the low cost and feasibility of the hardware features, a portion of the hardware performance monitor implemented for NUMAchine, a cache-coherent shared-memory multiprocessor, is presented.

To aid in multiprocessor research the hardware monitor should collect enough data to obviate the need for some detailed architectural simulations, which often run too slowly to collect statistically reliable data. Additionally, the monitor should assist machine and workload characterization for higher level simulation models; typical characteristics are the latency of memory read operations under various conditions or the probability of a read versus a write. These characteristics can be used to form powerful abstractions

because they permit rapid testing of ideas using faster, high-level simulations. Also, the measurements are useful to verify the results from such simulations.

A hardware performance monitor has a special appeal to software developers who wish to do software tuning. In this application, the goal of the hardware is not to replace software tuning tools, but to improve their instrumentation so that data collection is done nonintrusively and in real time. Additionally, the collected data should be fine-grained enough to identify, as best as possible, the true bottleneck to be optimized.

In the development of performance-monitoring hardware, a secondary objective is to make performance data accessible to a running program. By doing this, it is hoped that new research into software that can make runtime decisions to improve performance will be made possible. To achieve this goal, it is important to move the monitoring features as close to the processor as possible so that performance data queries can be satisfied quickly.

## 1.3 Contributions

The contributions made by this work can be summarized as follows:

- comprehensive list of 23 performance measurements for multiprocessors,
- extending informing memory operations to include an external *TRIGGER* pin,
- using numerous counters selected by a *PhaseID* register to partition performance data,
- returning the memory state with memory responses to estimate coherence overhead,
- definition, detection, and measurement of invalidation and ownership misses, and
- cache conflict measurements, fine-grained timing hardware, and quantifying latency-hiding mechanisms on performance comprise other minor contributions.

Although the contributions are made in the context of multiprocessors, many of the measurements and hardware mechanisms can be applied to uniprocessors as well. The difference is that coherence misses are eliminated and miss latency is more regular (*i.e.*, the effects of contention and maintaining coherence are reduced or eliminated) in uniprocessors.

## 1.4 Overview

This thesis is organized as follows. Chapter 2 describes previous work in both software and hardware monitoring systems. Chapter 3 proposes a number of measurements that can be made, including both new ideas and those borrowed from previous research, which are consistent with the main objectives. As an implementation example, a portion of the NUMAchine performance monitor is shown in Chapter 4. Finally, conclusions are given in Chapter 5.