Integrating Complete-System and User-level Performance/Power Simulators: The SimWattch Approach

Jianwei Chen, Michel Dubois and Per Stenström

Technical Report Number (ex: CENG 02-07)

Department of Electrical Engineering – Systems
University of Southern California
Los Angeles, California 90089-2562
213-740-4465

Integrating Complete-System and User-level Performance/ Power Simulators: The SimWattch Approach

Jianwei Chen[1] **Michel Dubois**[1]

[1]Department of Electrical Engineering - Systems
University of Southern California,
Los Angeles, CA 90089-2562, U.S.
213 740-4475
E-mail: {jianwei, dubois}@paris.usc.edu

Per Stenström[2]

[2]Department of Computer Engineering
Chalmers University of Technology,
SE-412 96 Gothenburg, Sweden
+46 (0)31 772 1761
E-mail: pers@ce.chalmers.se

Abstract

Most applications driving the advancements in microarchitecture and memory system research have a non-negligible interaction with the operating system. Yet, most architectural investigations are based on user-level simulators in which operating system activity is not modelled. This has motivated us to design SimWattch, a microarchitectural modeling infrastructure. SimWattch is based on Simics – a system-level simulation tool – and Wattch (SimpleScalar extended with power modeling) – a flexible user-level simulation tool. As a result, it can analyze performance and power dissipation in microarchitectures at the cycle level for complex workloads running on commodity operating systems.

In this paper, we present the design issues we had to resolve in integrating a system-level with a user-level simulator. We then use SimWattch to identify the type of errors a user-level simulator typically does when predicting performance and power dissipation while omitting operating system activity. Our results, which are based on applications such as SPEC95, SPEC JVM98, and TPC-B, show that the error can be large for commercial workloads such as TPC-B, in which IPC rate is overestimated by more than 20% and total energy used is underestimated by more than 100%. Although errors are lower for other benchmarks, they are still surprisingly high.

This paper demonstrates 1) that O/S activity must be modelled for accurate microarchitecture evaluations 2) that modelling O/S activity is feasible and its cost is reasonable, and 3) that complex and realistic microarchitecture evaluations including O/S activity can be done in reasonable simulation times today.

Keywords: computer architectures, simulation, operating systems, database systems, performance and power dissipation analysis.

1 Introduction

The applications that drive high-end computing have moved from scientific/engineering to information processing applications, such as database, decision support, and web search engines. In analyzing performance as well as power dissipation tradeoffs in computer architecture research, one should therefore reflect this change. Many studies have contrasted the characteristics of scientific/engineering with commercial applications and found noticeable differences. For example, Maynard *et al.* [19] showed that commercial applications are typically more challenging with respect to their interaction with the memory system and branch prediction mechanisms. One distinctive factor is that they typically have more operating system interactions than scientific/engineering applications. By contrast, scientific/engineering applications typically spend a very small fraction of their execution in system calls. As a result, it has been an accepted practice in computer architecture research to completely disregard the impact of operating system code on architectural decisions.

Over the years, the development of simulation tools to aid in performance and power dissipation tradeoff analyses of architectural decisions has also reflected the change in application focus. For a long time, user-level simulators, in which system calls are executed inside the simulator, have been the primary vehicle in architectural research. Several tools such as Asim [9], MINT [27], Rsim [15], Shade [8], and SimpleScalar [5] belong to this group. Recently, to satisfy the requirement for fast and flexible power evaluation tools, architectural-level power estimation tools like Wattch [4], SimplePower [28] and Architectural Power Evaluation [6] have been introduced. SimpleScalar, on which Wattch is based, is the most widely used user-level simulator because of its flexibility in modeling a wide range of microarchitectural design points. Despite this fact, since it ignores the effect of the operating system, it can make gross errors for applications from the information processing domain.

The development of complete system simulators has been directly motivated by the inability of user-level simulators to target complex workloads. Although complete system simulation tools are extremely large and complex, their benefits are diverse and significant [13][25]: evaluation of hardware design, development of operating system, and performance tuning of workloads. Complete system simulation approaches such as Simics [17][18] and SimOS [21] can functionally model the execution of applications with operating system interaction on the instruction-set architectural abstraction level. By choosing that abstraction level, the simulation speed is fast enough to measure execution statistics that normally can be captured by hardware counters in the context of realistically sized workloads. The SimOS project further demonstrated that it is possible to connect detailed microarchitectural simulators to predict the performance of microarchitectural features. SimOS is an environment for analyzing performance of architectural and software design alternatives of computer systems [13][21]. It can boot IRIX 5.3 or DEC UNIX [3] and run realistic workloads. SimOS contains three microarchitecture models to tradeoff simulation speed and detail. Recently, the capability of extending a MIPS microarchitectural model with power models has also been demonstrated [11]. However, none of these microarchitectural models provides the flexibility of SimpleScalar (or Wattch) to easily change microarchitectural resources. Additionally, SimOS provides flexible data collection and classification mechanisms, assisting the users to clearly and comprehensively understand simulation statistics. A simulator based on SimOS and called PharmSim is also being currently developed to evaluate microarchitecture in the context of complete system simulation [7].

This paper introduces SimWattch, a simulator that currently estimates performance and power consumption of a range of out-of-order issue superscalar microprocessors in a complete machine simulation environment. It is capable of supporting analysis of power-efficient microarchitecture, application, compiler and operating system design decisions. It does this by combining the powerful system-level support provided by Simics with the cycle-level microarchitecture timing and power estimations provided by Wattch.

SimWattch will especially fill the important gap of supporting microarchitectural research to improve energy efficiency. For example, recent studies targeting issue logic [10], instruction dispatch logic [16], pipeline mechanisms [2] have used user-level simulation approaches; thus, they have not been able to study the effects of workloads that rely on operating system support. Being based on SimpleScalar, the most widely used simulation tool for microarchitectural research, makes SimWattch a very attractive tool in the microarchitectural research domain.

The chief contribution of this paper is the approach by which we managed to integrate two very different simulators built independently and widely used in industry and academia: a user-level simulator (SimpleScalar/Wattch) and a complete-system simulator (Simics). By contrast, in SimOS [21], the integration of the two simulations is very tight and was planned when they were both designed. In SimOS, the microarchitecture simulator interacts with the system-level simulator at each instruction fetch, which may cause a lot of overhead. In SimWattch we take advantage of Simics' high-speed instruction execution by letting it produce an instruction trace at full speed. This instruction trace is buffered in a FIFO queue and consumed by Wattch at a slower pace. This mechanism introduced several non-trivial synchronization issues such as how to support speculative execution. We present in the next two sections the complete design of SimWattch and the fashion in which we solved these problems.

A second contribution in this paper is to use SimWattch to analyze the kind of errors made by user-level simulators which ignore the effect of operating system code in analyzing performance and power dissipation tradeoffs. By measuring miss rates, IPC, energy, power, and occupancy rates of various processor resources in a superscalar processor executing applications such as SPEC95int, SPEC JVM98, and TPC-B running on MySQL, we show that the errors can be very high for some metrics. For example, the relative error in predicting IPC for TPC-B using a user-level simulator is typically 20% whereas the energy used can be underestimated by more than 100%. Significant errors are also observed for power and energy per instruction, even in the context of SPEC95. This error analysis is presented in Section 4. We conclude the paper in Section 5 and expose our future research plans.

2 Overview of SimWattch

The overall structure of the SimWattch simulation environment is illustrated in Figure 1. Basically, SimWattch contains three main components: Simics, the complete system simulator; a modified version of Wattch, the cycle-level microarchitectural performance and power simulator; as well as the SimWattch Control Interface (SCI). On top of SimWattch, the Solaris 8 operating system and all workloads that rely on such operating system support can execute.

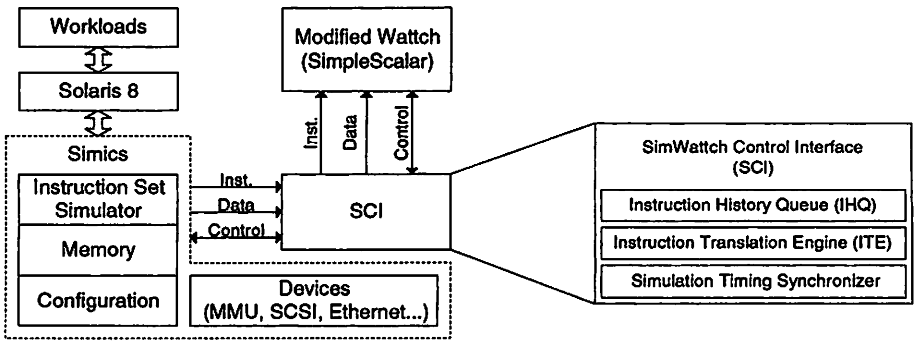

Workloads
Solaris 8
Simics
Instruction Set Simulator
Memory
Configuration
Devices (MMU, SCSI, Ethernet...)
Inst.
Data
Control
SCI
Modified Wattch (SimpleScalar)
Inst.
Data
Control
SimWattch Control Interface (SCI)
Instruction History Queue (IHQ)
Instruction Translation Engine (ITE)
Simulation Timing Synchronizer

Figure 1: The architecture of SimWattch.

Simics is a complete system simulator that functionally simulates various microprocessors and peripheral devices. It can run unmodified operating systems through its highly tuned instruction-set simulator that interacts with its memory and device models needed to accommodate an operating system. In our framework, Simics is employed to dynamically generate instruction traces, including operating system activity, at high speed and supply sufficient information to the microarchitecture level processor and memory simulator which is responsible for detailed modeling of performance and power at the cycle level.

The microarchitecture level performance and power simulation engine of SimWattch is built on Wattch [4], a power modeling extension of SimpleScalar. The SimpleScalar toolset models a range of detailed, dynamically scheduled processors with multiple-level memory hierarchies, speculative execution, and state-of-the-art branch prediction at the cycle-level. Although a few significant changes are made to SimpleScalar for supporting the SPARC-V9 instruction set, a goal has been to maintain compatibility with the SimpleScalar toolset as much as possible through providing an interface to Simics that will make it possible to upgrade SimpleScalar's future extensions and to retain the advantages of SimpleScalar's flexible microarchitecture configuration, and well documented and structured designs [1].

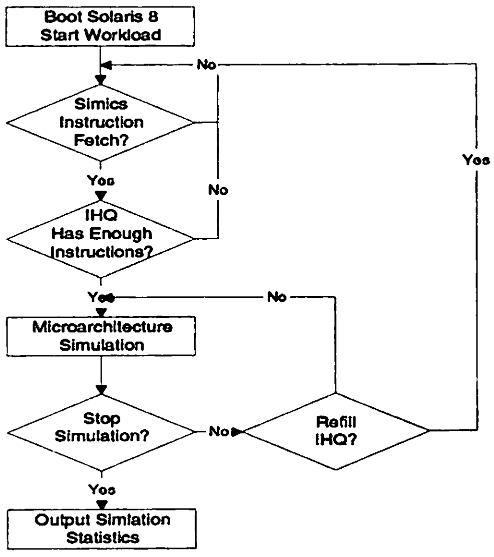

Boot Solaris 8
Start Workload
No
Simics Instruction Fetch?
Yes
No
Yes
IHQ Has Enough Instructions?
Yes
No
Microarchitecture Simulation
Stop Simulation?
No
Refill IHQ?
Yes
Output Simlation Statistics

Figure 2: Simulation flow of SimWattch.

The SimWattch Control Interface (SCI) implements the interface between Simics and Wattch. SCI is responsible for three tasks: First, it buffers the committed instructions executed on Simics; second, it serves memory accesses issued by the microarchitecture simulation module; and finally, it dynamically translates SPARC-V9 instructions into the native instruction set of SimpleScalar. Before explaining its design and implementation, we at first provide a brief description of the SimWattch work flow, which is displayed in Figure 2.

The entire simulation procedure can be divided into three main stages. The aim of the first stage ("Boot Solaris 8" in Figure 2) is to prepare the simulation environment for the following detailed performance and power simulation. This stage only involves Simics and takes advantage of its high functional simulation speed which provides a slowdown of just two orders of magnitude. It functionally simulates a few billion instructions to boot Solaris 8, runs workloads until some point where we are interested to do detailed analysis and generates a checkpoint containing the entire simulated machine state. In our experiment, this stage takes around ten minutes on a Sun SunBlade 1000 workstation. Just like in SimOS [13], we can skip the above steps by loading the checkpoint in a later simulation session.

In the second stage ("Simics Instruction Fetch" in Figure 2), the Instruction History Queue (IHQ) is filled with an instruction trace from Simics. For each committed instruction in Simics, the instruction word, its PC, and its associated operands are stored in the next free slot of the IHQ. The above process keeps going until the IHQ is full.

In the third stage ("Microarchitecture simulation" in Figure 2), the microarchitecture simulation module starts cycle-level performance and power simulation. It retrieves SPARC V9 instructions from the IHQ and translates them to an extended set of the native Simplescalar ISA (PISA) to model the SPARC V9 ISA. It then models its execution on the microarchitecture model.

Synchronizing Simics with SimpleScalar, preserving memory semantics under speculation as well as exception handling pose interesting challenges. These issues are discussed in the next section.

3 Design Decisions in SimWattch

3.1 Synchronization Between Simics and SimpleScalar

When integrating a microarchitectural simulator with a system-level simulator, the most obvious choice would be to let the microarchitecture simulator act as a master that lets the system-level simulator carry out memory accesses. This is the approach taken in most simulators. However, in Simics, the memory access path, which involves address translation, is heavily optimized so that it only consumes between 10-50 instructions in the common case of a successful address translation.

To take advantage of the optimized memory access path in Simics, a key design decision we made was to let Simics act as the master by executing instructions in advance and inserting committed instructions into the Instruction History Queue (IHQ) as shown in Figure 3. Apart from synchronizing Simics with SimpleScalar, the IHQ also helps to match the speed between them.

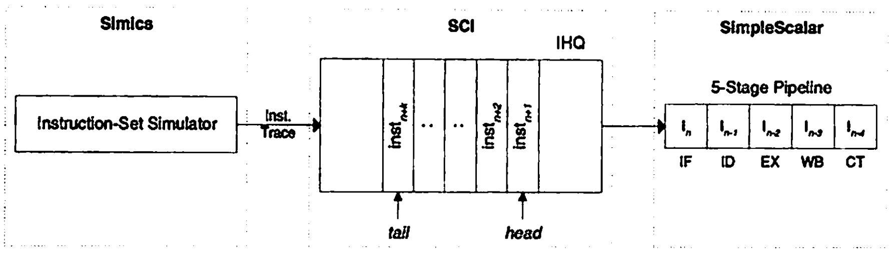


Figure 3: Synchronization between Simics and SimpleScalar

3.2 Speculative Execution and Exception Handling

SimpleScalar supports speculative execution. The fact that Simics executes and commits instructions, without speculating along a mispredicted path, introduces a number of challenges to make SimWattch support speculative execution along correct as well as mispredicted paths.

As long as SimpleScalar executes along the correct path, Simics will always be in advance and will have already inserted the correct instructions in IHQ. Conversely, when SimpleScalar executes along a mispredicted path, it will only find the instruction in IHQ if it has been previously visited on the correct path in which case the instruction address is used to locate it in IHQ. Otherwise, we let SCI access the memory model in Simics directly to retrieve the instructions needed by the speculation. This may lead to an address translation exception, e.g., a TLB miss or a page fault. We deal with such exceptions on mispredicted paths by signalling misprediction and squashing the execution of all instructions in the pipeline by injecting No-ops. This approach will neither pollute the caches nor the TLB.

Another issue concerns how to preserve memory semantics. Since Simics executes in advance special precaution have to be taken when SimpleScalar executes load and store instructions. As long as SimpleScalar speculates along the correct execution path, Simics has already committed all these speculated instructions and updated the memory state accordingly. As a result, store instructions issued by SimpleScalar can be handled silently inside SimpleScalar. On the other hand, loads issued by SimpleScalar must reflect the memory state as it was when the load instruction was executed. Since Simics executes in advance, it may have modified the memory state. To guarantee that SimpleScalar loads the correct value, the source operand of the load is inserted in the IHQ along with the load instruction and SimpleScalar simply gets the data from there.

When SimpleScalar executes along a mispredicted path, the values associated with store instructions will never be committed to memory. Load instructions should however return the value of the latest store which either corresponds to the checkpointed memory state at the beginning of the speculation or to a subsequent store to the same address along the mispredicted path. Because the effect of feeding mispredicted load instructions with incorrect data is expected to be small, we let a load instruction that executes along a mispredicted path read directly from memory assuming that the memory state is up-to-date most of the time.

Exception processing leverages on the support for speculative execution. It can be viewed as an unexpected procedure call and causes a change of the current instruction flow. In SimWattch, a trap is detected in the instruction dispatch stage. We then treat it as a branch misprediction besides preserving and updating the simulated machine states [23], which is not needed when dealing with a normal branch misprediction. After that, the instruction flow resumes from the first instruction of the trap handler.

3.3 Architecture Mapping

Since Simics does not implement the PISA instruction set, we had to make an architecture mapping between SPARC V9 and PISA. Table 1 displays the major differences between the two architecture models.

	SPARC-V9	PISA
Instruction Size	32-bit	64-bit
Address Space	64-bit	32-bit
Register Window	Yes	No
INT Registers	32 GPR (64-bit)	32 GPR (32-bit)
FP Registers	32 Single-Precision (32-bit) 32 Double-Precision (64-bit) 16 Quad-Precision (128-bit)	32 Single-Precision (32-bit) 16 Double-Precision (64-bit)

Table 1: Differences in the register model of SPARC-V9 ISA and SimpleScalar PISA.

Each processor architecture has its own register organization using a convention that may be different from others. Thus, to run unmodified SPARC-V9 instructions on the SimpleScalar architecture, the SPARC-V9 register state must be emulated by SimpleScalar. We extended the SimpleScalar register model to match the floating-point and control/status registers in SPARC V9. We also implemented the register window containing 136 integer registers in SimpleScalar. In addition, the 32-bit SimpleScalar address space is extended to 64-bit to emulate the SPARC-V9 memory organization.

Instruction translation was simplified significantly because both architectures support the same addressing modes and has a similar instruction format. In SimWattch, mapping the layout of each SPARC-V9 instruction word to PISA is dynamically conducted by the SimWattch Instruction Translation Engine. At first, it decomposes a SPARC-V9 instruction into several parts, and then fills them into the corresponding fields of the PISA instruction format after a few minor changes. In our implementation, each generic SPARC V9 instruction accessing no more than three registers is translated into a single PISA instruction.The benchmarks we have run so far have been successfully translated by the ITE.

3.4 Power Models

The dynamic power consumption of CMOS microprocessors is in Wattch [4] modeled as

$$P_d = CV_{dd}^2 \alpha F$$

where C is the load capacitance, V_{dd} is the supply voltage, and F is the clock frequency. The activity factor, α, is a fraction between 0 and 1 indicating how often clock ticks lead to switching activity on average.

The power models in Wattch [4] assume that the main processor components are arranged into four groups: array structures, CAM, combinational logic and wires, and clocking. Based on the above equation, the physical character of circuits and process technology, different C, V_{dd}, α and F are modeled for various component groups to calculate their power consumption. On each cycle of the simulation of an application, the power models for each active unit will be invoked to calculate and record the energy consumed. In addition, Wattch implements different conditional clocking styles: (1) no conditional clocking; (2) simple conditional clocking; (3) aggressive, ideal conditional clocking: zero power consumption when a unit is turned off; and (4) an aggressive, non-ideal conditional clocking: power of active units is scaled linearly with port or unit usage, and unused units still dissipate 10% of the maximum power rather than zero. In this paper, we assume a 0.18 micron process technology at 600MHz, and aggressive, non-ideal conditional clocking style is employed for all results.

3.5 Validation

The critical issue in validating the SimWattch infrastructure has been to make sure that the instruction trace presented to SimpleScalar is the same as the one issued by Simics and that the extension of the PISA instruction set to implement SPARC-V9 semantics is correct. We have validated the infrastructure using several approaches. First, we have measured the instruction mixes, i.e. the fraction of dynamically executed instructions in each category, for the executed instructions on Simics and for the retired instructions on the modified version of Wattch for the four applications to be presented later and found that they were the same. Second, as far as the implementation of SPARC-V9 on SimpleScalar we analyzed in detail that the instruction dependencies are maintained correctly in the pipeline by analyzing the instructions from a set of small test programs cycle by cycle. As for the correctness of the power model, we rely on the validation of Wattch [4].

4 O/S Effects on Micro-architecture Simulations

Intuitively, it would seem important to include operating system code in the evaluation of microarchitectures. There are three reasons for this. First we expect that operating system code has a distinctive behavior different from user code, and, therefore, in applications relying heavily on operating system services – such as commercial workloads – we would expect that the overall microarchitecture performance is affected by both types of code, not just user code. Second, we expect significant interactions between O/S and user codes as they dynamically execute in turn: Right after returning from an O/S call, it is likely that the state of the microarchitecture – including pipeline and caches – is markedly different from the state of the micro-architecture before the call was made, resulting in pollution effects inside the microarchitecture. This pollution effect may modify the sequence of simulated events in the user code for some time and impact measurements. Third, the timing of activities scheduled in an O/S call may affect the behavior of the micro-architecture. For example the timing of I/O responses may affect the contents of caches, the timing of future O/S calls and the interference of O/S code activities with user code activities in the micro-architecture.

To verify this intuition and justify the need for including O/S code in simulations, we have performed several simulation evaluations using SimWattch. The goals of these evaluations are 1) to compare the effectiveness of microarchitectures for SPEC applications and commercial workloads, 2) to observe errors in micro-architecture simulation due to O/S code omission, and 3) to demonstrate that SimWattch can be used to conduct a very detailed and flexible performance and power analysis of a complex micro-architecture in the context of a complete system simulation.

4.1 Architecture Model

Table 2 gives the default microarchitecture parameters of a state-of-the-art 4-issue superscalar processor studied in this paper. Particularly, this architecture implements out-of-order issue and execution, based on a

Register Update Unit (RUU) [22]. RUU is a unified instruction window, which uses a reorder buffer to automatically rename and hold the results of pending instructions. Unless otherwise stated, the results in this paper are collected based on this default configuration.

Parameter	Value
RUU size	128 instructions
LSQ size	8 instructions
Fetch Queue size	4 instructions
Fetch, Decode, Issue and Commit width	4 instructions/cycle
Issue Mechanism	out-of-order
Functional Units	4 INT ALUs, 4 FP ALUs 1 INT Multiply/Divide 1 FP Multiply/Divide
Branch Predictor	Combined, Bimodal 4K table, 2-Level 1K table, 10-bit history table 4k Chooser
BTB	512-entry, 4-way
Mispredict Penalty	7 cycles
L1 Instruction Cache	128K, direct-mapped, 32B blocks, 1 cycle latency
L1 Data Cache	128K, 4-way (LRU), 32B blocks, 1 cycle latency
L2 Cache	Unified, 1M, 4-way (LRU), 64B blocks, 3 cycles latency
Memory	100 cycles first chunk, 30 cycles inter chunk
I-TLB/D-TLB	128-entry, fully associative, 30-cycle miss penalty

Table 2: Microarchitectural default parameters.

4.2 Benchmarks

We have picked four programs from SPECint95, SPEC JVM98 and TPC-B running on the MySQL database engine. The four programs run on top of Solaris 8. SPEC95 is a suite of compute-intensive benchmarks. They stress the performance of the processor, the memory hierarchy, and the compiler. SPEC JVM98 not only evaluates CPU (integer and floating-point), cache and memory architectures but also stresses the efficiency of JVM, the JIT compiler, and the operating system implementations. SPEC JVM98 contains more operating system activities than SPEC95. In addition, a database workload, TPC-B running on the MySQL database engine is analyzed. TPC-B is an official OLTP benchmark, and has similar behavior to its successor, TPC-C. MySQL is a popular open source database engine supporting many websites, datawarehouses, business applications, and logging systems (e.g. in Yahoo, Finance, MP3.com, and Motorola). Database workloads are known to heavily depend on operating system support and have a radically different behavior from scientific and engineering workloads [19]. Thus, SPEC95, JVM98 and TPC-B are interesting candidates in evaluating a complete performance and power simulation environment.

A detailed description of the benchmarks is shown in Table 3. We compiled them for the SPARC-V9 ISA using GCC 3.0 with -O2 optimization flag. However, *_201_compress* runs in interpreter mode. To evaluate the essential execution behavior of the workloads, we simulate 100 million instructions after skipping the initialization phase of each program.

Category	Benchmark	Input Set	Description
SPECint95	126.gcc	protoize.i	GNU C compiler
	129.compress	30000 q 2131	UNIX compression utility
JVM98	_201_compress	s1	A java port of the 129.compress
Database	TPC-B (MySQL)	40 branches	Database management systems

Table 3: Benchmarks, input set, and description.

In our simulations all statistics are collected in two modes: Overall and User. Overall numbers cover the entire execution of the 100M instructions including O/S and user codes. User numbers are collected for user instructions only: O/S instructions in the 100M instructions are filtered out by Simics, which means only user instructions are fed into the IHQ and simulated in the microarchitecture simulation module. Since the microarchitecture is not affected by instructions other than user instructions, the numbers reported under User would be the same numbers as reported by a user-level simulator ignoring O/S code.

4.3 Performance Characteristic

4.3.1 Instruction Mixes

Figure 4 and Table 4 show the basic instruction mix of the four workloads. First, the fraction of user instructions are given in Figure 4. As expected, *TPC-B* has the least user-level activity. A majority of its work (50.9%) is done by the operating system. By contrast, engineering workloads such as *126.gcc* and *129.compress* exercise the operating system very little. And user instructions make up more than 97% of its total execution. Surprisingly, even *_201_compress* has very little operating system activity. Table 4 shows that the inclusion of operating system code has little or no impact on the instruction mix of engineering workloads whereas the instruction mix of *TPC-B* is somewhat affected.

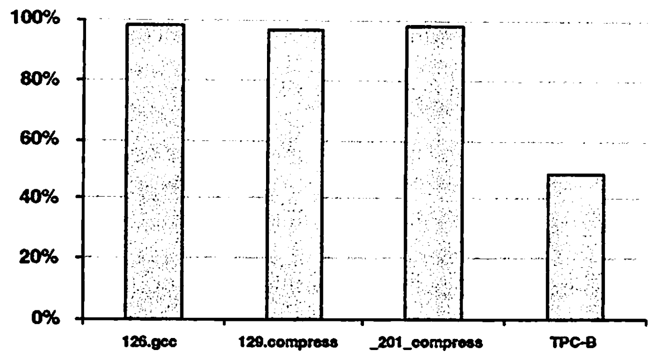


Figure 4: Percentage of user instructions.

	126.gcc	129.compress	_201_compress	TPC-B
Load	17.9/17.9	12.6/12.5	22.9/23.0	17.6/16.9
Store	4.23/4.04	7.66/7.71	7.97/7.96	9.78/7.16
Branch	22.3/22.3	10.6/10.5	13.6/13.6	19.2/21.1

Table 4: Instruction mix in percent (Overall/User).

4.3.2 Miss Rates

Table 5 shows the miss rates in caches and TLBs. They qualitatively match the results reported in [19].The overall miss rates in caches and TLBs are consistently higher than the corresponding miss rates in user mode. The difference is particularly large for TPC-B and is due to the pollution of caches and TLBs by O/S code execution.

	126.gcc	129.compress	_201_compress	TPC-B
L1 ICache	1.31/1.16	0.04/0.00	0.07/0.01	6.13/4.52
L1 DCache	0.38/0.27	3.18/3.14	3.47/3.37	0.91/0.29
L2 Cache	5.67/4.03	2.50/1.12	3.58/2.58	1.76/1.04
ITLB	0.02/0.01	0.01/0.00	0.01/0.00	0.41/0.34
DTLB	0.06/0.01	0.14/0.06	0.13/0.01	0.94/0.02

Table 5: Cache and TLB miss rate in percent (Overall/User).

	126.gcc	129.compress	_201_compress	TPC-B
Branch Direction Prediction Accuracy	88.98/89.01	91.35/91.41	93.43/93.46	91.21/91.26
Branch Address Prediction Accuracy	82.49/82.81	90.80/91.08	92.60/92.98	72.39/78.82

Table 6: Branch prediction accuracy in percent (Overall/User).

Table 6 gives branch prediction accuracy statistics. We see little difference between overall and user codes, except for TPC-B where frequent O/S traps foils the branch address prediction hardware.

Miss rate numbers in the memory hierarchy showing the need to include O/S code in commercial workloads are not new, as they can be derived easily with Simics or SimOS [3, 7, 13, 18, 19]. In the context of SimWattch, we can further evaluate the effect of O/S code execution on the microarchitecture efficiency and power dissipation.

4.3.3 Microarchitecture Efficiency

Figure 5 shows the IPC (instructions per cycle) of the four workloads in the default hardware configuration. The overall IPC varies from 2.3 for *129.compress* to 0.68 for *TPC-B*, as it is well known that engineering workloads tend to have instruction level parallelism that is easier to exploit. In general, the overall IPC is lower than the user IPC. This is expected in *TPC-B,* which has a large fraction of OS instructions. The effect of O/S code on IPC in *129.compress* and *_201_compress* is a bit surprising, given the small amount of O/S activity in these applications, but can still be explained by the pollution of the caches and microarchitecture by OS code execution.

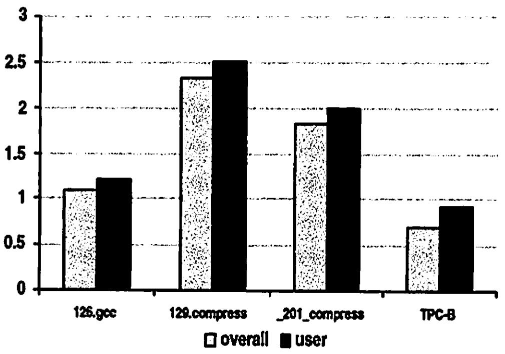


Figure 5: IPC results.

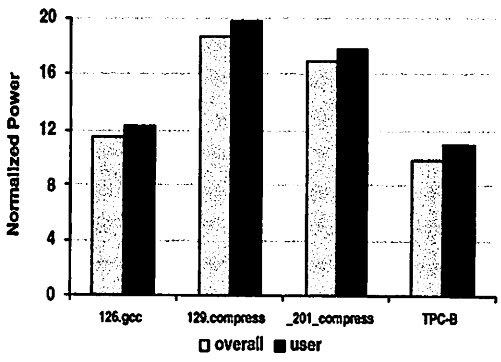


Figure 6: Power results.

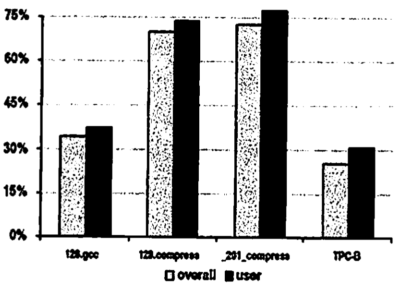


Figure 7: IFQ occupancy.

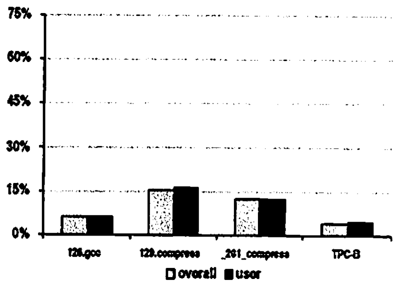


Figure 8: RUU occupancy.

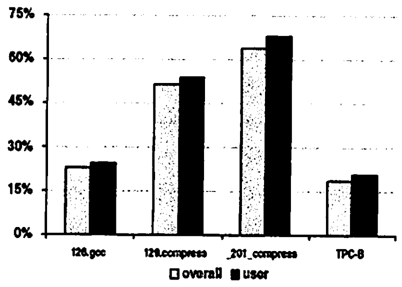


Figure 9: LSQ occupancy.

Figures 7, 8 and 9 show the IFQ (instruction fetch queue), RUU (register update unit) and LSQ (load store queue) occupancy rate for the four workloads. Occupancy rates are related to IPC: high occupancy means adequate utilization of hardware resources, which eventually leads to high performance. The occupancy rates of *129.compress* and *_201_compress* are much higher than *126.gcc* and *TPC-B*. We see again that the simulation of user code only overestimates resource utilization.

4.4 Power and Energy

Another important application for the SimWattch infrastructure is measuring power and energy dissipated. The power dissipated (see Figure 6) is directly related to IPC. As for the IPC, power is overestimated by simulating user code only instead of all the code.

Figure 10 shows the total energy consumed by the entire execution and by user code only. In the three engineering workloads little energy is spent outside of user code. However, in *TPC-B*, the overall energy is more than twice the energy spent in user mode. Obviously, it is impossible for a user-level simulator to investigate adequately the energy consumption of database applications. Figure 11 shows the amount of energy spent to execute each instruction. This metric removes the effect of the different number of instructions executed in Overall and User. Differences between User and Overall are less pronounced but still exist.

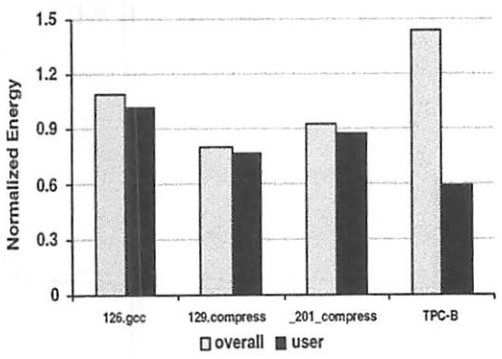


Figure 10: Total energy.

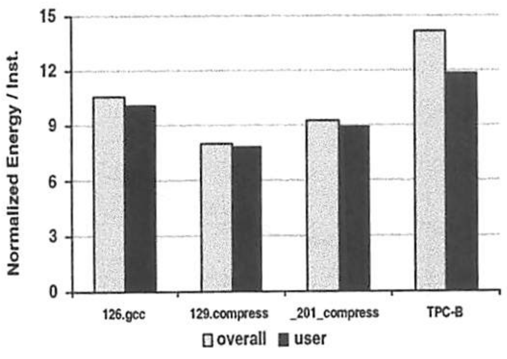


Figure 11: Energy / instruction.

4.5 Energy and Performance Tradeoffs

An important application of SimWattch is to study the impact of architectural changes on power and performance. In the following, we consider a range of implementations where we successively increase the issue width, which is always equal to the decode and commit width. We use the parameters of the default configuration, except that, to prevent integer ALUs from becoming the performance bottleneck, we use eight integer ALUs when the issue width is equal to eight. The rest of the microarchitectural resources are kept unchanged. As issue width increases, performance improves, but renaming, instruction window, register file and result bus consume more and more power.

Figures 12 through 15 show the trends of IPC and power, as the issue width increases from one to eight. *129.compress* reaps a significant performance gain from increasing the issue width. To support this level of performance, the power increases almost linearly with the issue width and the IPC. By contrast, *TPC-B* is not able to exploit wider issue widths because it does not have enough instruction-level parallelism which can be exploited by more aggressive superscalar design. Because IPC gains are so modest, power dissipation increases only marginally.

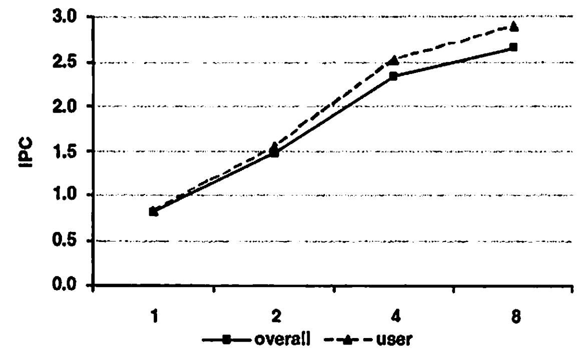


Figure 12: 129.compress IPC.

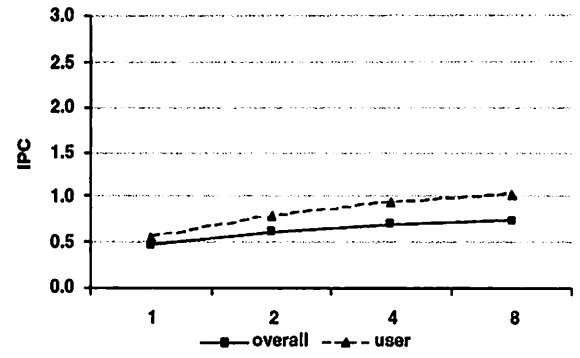


Figure 13: TPC-B IPC.

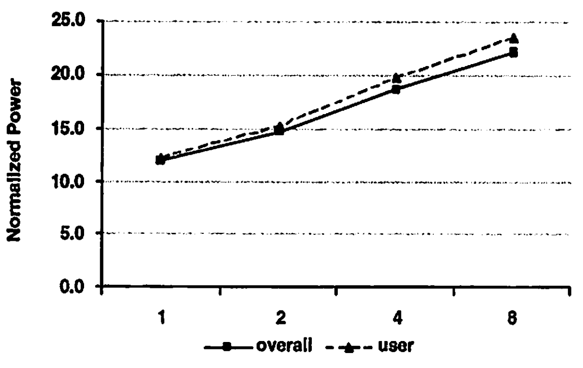


Figure 14: 129.compress power.

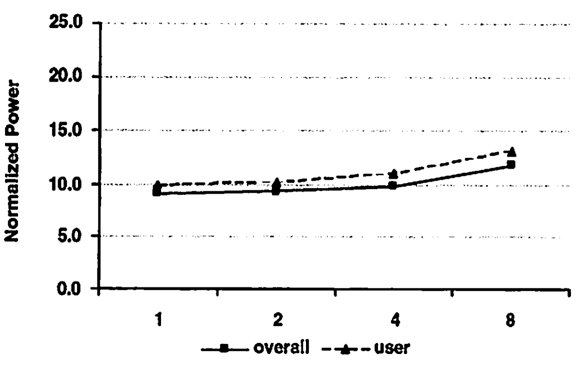


Figure 15: TPC-B power.

Figures 16 and 17 show that the total energy goes down as the issue width increases from one to four but, as the issue width increases further, the total energy consumed trends up. Figure 17 shows that the energy spent in user mode is a little more than a third of the overall energy in *TPC-B*. Thus one can expect to expand much more energy executing *TPC-B* than its user instructions would let us believe: The large majority of the energy is spent in O/S mode.

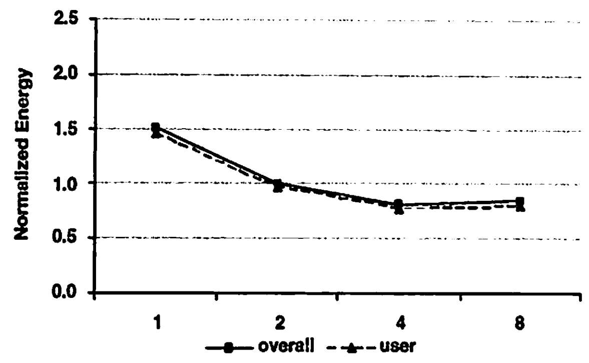


Figure 16: 129.compress total energy.

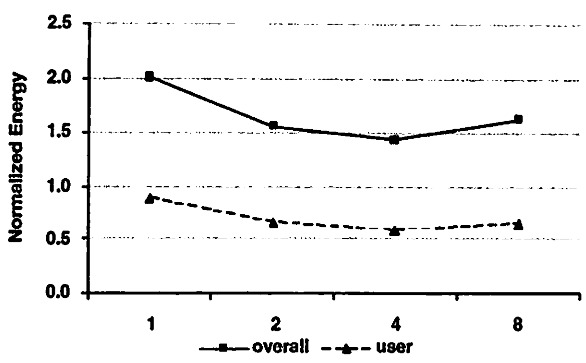


Figure 17: TPC-B total energy.

Figures 18 and 19 show the energy normalized to the number of instructions. The trend is the same as for the total energy. Energy per instruction is a fairer metrics than total energy for evaluations using user code only since it is independent of the number of instructions executed.

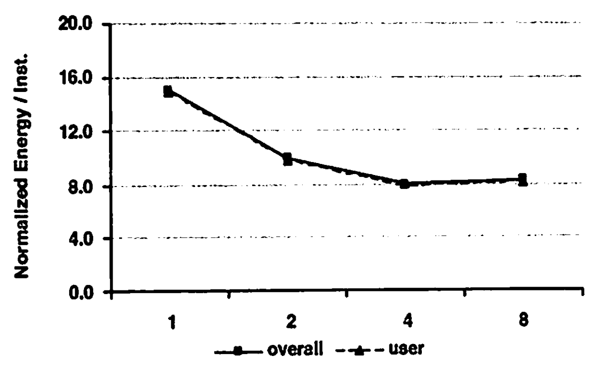


Figure 18: 129.compress energy / instruction.

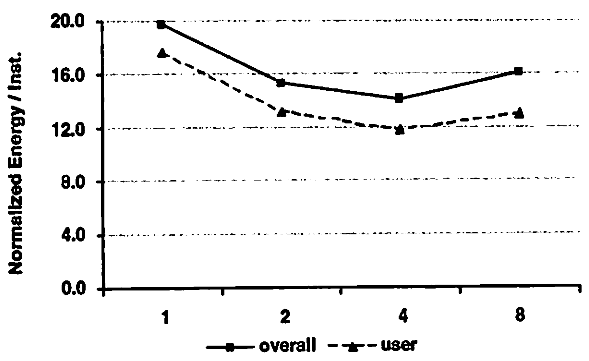


Figure 19: TPC-B energy / instruction.

When we look at figures, we see that simulating user code only can lead to significant overestimation of performance and significant underestimation of energy. Although the trends are similar in every case, the absolute values are sometimes completely off-fields, as in the case of IPC and energy for *TPC-B*.

Tables 7 and 8 show the relative errors caused by omitting O/S code in the simulations. The errors are computed as the absolute difference between overall and user values divided by user value. The error can be excessively large, especially for IPC and energy in the case of *TPC-B*. Surprisingly, relative simulation errors are non-negligible in the case of *129.compress* as well.

Issue Width	1	2	4	8
IPC	2.37	4.50	7.25	8.15
Total Energy	3.54	4.18	5.06	5.84
Energy per clock	1.81	3.37	5.35	5.58
Energy per Inst.	0.20	0.77	1.54	2.25

Table 7: Relative error due to O/S code omission in 129.compress (%).

Issue Width	1	2	4	8
IPC	17.45	22.17	25.91	27.21
Total Energy	127.15	136.72	145.55	151.96
Energy per clock	7.93	9.53	10.67	9.94
Energy per Inst.	10.04	14.33	17.01	17.36

Table 8: Relative error due to O/S code omission in TPC-B (%).

5 Conclusion and Future Work

In this paper, we have introduced SimWattch, a complete system simulation environment that can be used to conduct performance and power-oriented microarchitecture research or revisit existing techniques from a more complete perspective taking into account operating system interactions on the power dissipated in microarchitectural components. As a combination and extension of Simics and Wattch, it facilitates the analysis of a wider design space for computer architects, application, compiler and operating system developers. In addition, it explores a new approach to cost-effective simulator design.

We have reported on performance and power measurements made with SimWattch to motivate the inclusion of O/S code in microarchitecture simulations. To this end we measured both User and Overall statistics. The performance and energy effects of O/S code execution were surprisingly large in engineering workloads such as SPEC95, even if these workloads execute very little O/S code. IPC is broadly overestimated by ignoring O/S code, by more than 20% in *TPC-B*. Likewise, energy is underestimated by large amounts, especially for commercial workloads. Simulations ignoring O/S code are generally too optimistic and should be seen as a lower bound on energy and an upper bound on performance.

SimWattch is still in a preliminary development phase and there is room for improvement. Accuracy is a serious concern of existing architectural-level power analysis tools. First, current power models must be improved to factor in leakage power because this effect is becoming more important as we shrink the feature size further. Second, more powerful and accurate data collection and classification mechanism can help us have a better and clearer understanding of different system/user behaviors. Finally, the microarchitectural simulation speed of SimWattch is roughly 20k instructions/second on a SunBlade 1000 workstation running Solaris 8. This compares with a simulation speed of about 100k instructions on the same machine for SimpleScalar in user mode. We believe that the improved accuracy is well worth the effort of including O/S code. In any case, SimWattch's speed could be further improved through optimizing instruction translation and removing some of the function overlap between Simics and Wattch.

Extensions to the simulator are topics of future research. So far, SimWattch models microarchitectures. To fully exploit a complete system simulation tool such as Simics, power models of other modules in the system, such as disks, should be added. Additionally, research on power-efficient multiprocessor design is still in its infancy although its importance begins to attract more and more attention. We believe SimWattch can be an ideal simulation tool for power/performance studies on such topics.

Acknowledgments

We would like to thank STINT - The Swedish Foundation for International Cooperation in Research and Higher Education under the MECCA project for financial support. This work was also partly supported by NSF Grant CCR-0105761. We also thank Jim Nilsson who has been instrumental in giving feedback on the project and help with setting up the benchmarks.

References

[1] T. Austin, E. Larson and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling", *IEEE Computer*, Feb. 2002, pp. 59-67.

[2] R. Bahar and S. Manne, "Power and Energy Reduction Via Pipeline Balancing", in *Proc. 28th Ann. Int. Symp. Computer Architecture*, 2001, pp. 218-229.

[3] L. A. Barroso, K. Gharachorloo and E. Bugnion, "Memory System Characterization of Commercial Workloads", in *Proc. 25th Ann. Int. Symp. Computer Architecture*, 1998, pp. 3-14.

[4] D. Brooks, V. Tiwari and M. Martonosi, "Wattch: A Framework for Architectual-Level Power Analysis and Optimizations", in *Proc. 27th Ann. Int. Symp. Computer Architecture*, 2000, pp. 83-94.

[5] D. Burger and T. M. Austin, "*The SimpleScalar Tool Set, Version 2.0*," University of Wisconsin-Madison Computer Sciences Department Technical Report #1342, 1997.

[6] G. Cai and C. H. Lim, "Architectural Level Power/Performance Optimization and Dynamic Power Estimation", in *Proc. Cool Chips Tutorial*, in conjunction with MICRO32, Nov. 1999, pp. 90-113.

[7] Harold W. Cain, Kevin M. Lepak, Brandon A. Schwartz and Mikko H. Lipasti, "Precise and Accurate Processor Simulations", *Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads, HPCA-8*, Feb. 2002.

[8] R. Cmelik and D. Keppel, "*Shade: A Fast Instruction-Set Simulator for Execution Profiling*", Sun Microsystems Labs Technical Report TR-93-12, 1993.

[9] J. Emer, P. Ahuja, E. Borch, A. Klauser, C. Luk, S. Manne, S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa and T. Juan, "Asim: a performance model framework", *IEEE Computer*, Feb. 2002, pp. 68-76.

[10] D. Folegnani and A. Gonzalez, "Energy-Effective Issue Logic", in *Proc. 28th Ann. Int. Symp. Computer Architecture*, 2001, pp. 230-239.

[11] S. Gurumurthi, A. Sivasubramaniam, M. J. Irwin, N. Vijaykrishnan, M. Kandemir, T. Li and L. K. John, "Using Complete Machine Simulation for Software Power Estimation: The SoftWatt Approach", *HPCA-8*, Feb. 2002, pp. 387-398.

[12] S. A. Herrod, "The SimOS Simulation Environment," http://simos.stanford.edu, Oct. 1997.

[13] S. A. Herrod, *"Using Complete Machine Simulation to Understand Computer System Behavior,"* Ph.D. thesis, Stanford University, Feb. 1998.

[14] http://www.mysql.com/

[15] C. Hughes, V. Pai, P. Ranganathan and S. Adve, "Rsim: simulating Shared-Memory Multiprocessors with ILP Processors", *IEEE Computer*, Feb. 2002, pp. 40-49.

[16] G. Kucuk, K. Ghose, D. V. Ponomarev and P. M. Kogge, "Energy-Efficient Instruction Dispatch Buffer Design for Superscalar Processors", in *Proc. ISLPED'01*, Aug. 2001, pp. 237-242.

[17] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, B. Werner, "Simics: A Full System Simulation Platform", *IEEE Computer*, Feb. 2002, pp. 50-58.

[18] P. S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson. F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström and B. Werner, "SimICS/sun4m: A Virtual Workstation," in *Proc. Usenix Annual Technical Conference*, Jun. 1998, pp. 119-130.

[19] A. M. Maynard, C. M. Donnelly and B. R. Olszewski, "Contrasting Characteristics and Cache Performance Technical and Multi-User Commercial Workload", in *Proc. ASPLOS'94*, Oct. 1994, pp. 145-155.

[20] T. Mudge, "Power: A First Class Design Constraint", *IEEE Computer*, Vol. 34, No. 4, Apr. 2001, pp. 52-58.

[21] M. Rosenblum, E. Bugnion, S. A. Herrod, and S. Devine. "Using the SimOS Machine Simulator to Study Complex Computer Systems", *ACM Transactions on Modeling and Computer Simulation*, Vol. 7, No. 1, Jan. 1997, pp. 78-103.

[22] G. S. Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers," *IEEE Transactions on Computers*, Vol. 39, No. 3, Mar. 1990, pp. 349-359.

[23] SPARC International, "The SPARC Architecture Version 9," 1992.

[24] Standard Performance Evaluation Corporation, http://www.spec.org, *SPEC 95, SPEC JVM98.*

[25] P. Strazdins, *"A Survey of Simulation Tools for CAP Project Phase III,"* Technical Report, Computer Science Laboratory, the Australian National University, Oct. 2000.

[26] Transaction Processing Performance Council. *TPC Benchmark B (Online Transaction Processing) Standard Specification*, 1990.

[27] J. Veenstra and R. Fowler, "MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors", in *Proc. MASCOTS'94*, 1994, pp. 201-207.

[28] N. Vijaykrishnan, M. Kandermir, M. J. Irwin, H. Kim and W. Ye, "Energy-Driven Hardware-Software Optimizations Using SimplePower", in *Proc. 27th Ann. Int. Symp. Computer Architecture*, 2000, pp. 95-106.

[29] K. Wilcox and S. Manne, "Alpha Processors: A History of Power Issues and A Look to the Future", in *Proc. Cool Chips Tutorial*, in conjunction with MICRO32, Nov. 1999, pp. 16-37.

Towards Virtually Addressed Memory Hierarchies

Xiaogang Qiu

Technical Report Number (ex: CENG 02-08)

Department of Electrical Engineering – Systems
University of Southern California
Los Angeles, California 90089-2562
213-740-4465

TOWARDS VIRTUALLY ADDRESSED MEMORY HIERARCHIES

by

Xiaogang Qiu

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(ELECTRICAL ENGINEERING)

August, 2000

Dedicated to my dear family.

Acknowledgments

First and foremost, I would like to thank my advisor Dr. Michel Dubois, for 5 years of support and advice. His valuable discussion, guidance and encouragement have led me to finally reaching this level.

I am grateful to Dr. Mary Hall and Dr. Jean-Luc Gaudiot who have been very kind to serve on my defense committee.

Many thanks to my colleagues at USC for their cooperation and friendship: Jaeheon Jeong, Kangwoo Lee, Koray Oner, Zhi Shi, Jianwei Chen, Adrain Moga, and Yongho Song.

I am indebted to Lucille Stivers and Brendan Char, who have always been very helpful to me.

Special thanks go to my friends in Los Angeles with whom the life of graduate school becomes such a pleasant experience. Thanks to my buddies, Kun Li, Aiguo Xie, Hai Zhang, Yuyong Zhang, and Xingang Zhang.

Many thanks to my family for their love and encouragement. My parents have provided so much generous love and have always been the ultimate support throughout my life. I would like to give my special thanks to my sisters, who have always been so patient and have always believed in me. I am also indebted to my parents-in-law, who have been so kind for helping us take care of my little baby in the difficult times.

I want to thank my wife, Weifang Xie for her dear love and support over years, and my son Robert, whose smile has always let me forget all the stress and geared me up with new energy and creativity.

Contents

List of Figures

List of Tables

Abstract

In current systems, virtual addresses are dynamically translated into physical addresses through a Translation Lookaside Buffer (TLB) accessed in parallel with the first level cache. Such a TLB does not scale well with processor speeds, physical memory sizes, and application data set sizes. Other observed trends in architecture are the migration of computations to memory (PIM) to fight the memory wall and the emulation in software of various memory functions. We believe that these trends will eventually drive architects of general-purpose systems to virtualize the memory hierarchy.

In this dissertation, we focus on memory systems where the TLB is moved down the memory hierarchy, away from the processor. We propose and evaluate a novel multiprocessor architecture called Virtual COMA(V-COMA), in which the virtual memory implementation is combined with the cache coherence protocol.

We also introduce new solutions to two critical problems plaguing systems with virtual memory hierarchies: synonyms and late memory traps.

To solve the synonym problem, we propose a Synonym Lookaside Buffer (SLB), which translates synonyms before virtual addresses are issued to the memory hierarchy. In contrast to TLBs, SLBs are very small and scale very well.

We then present an in-depth analysis of the trapping behavior of ILP (Instruction-Level Parallelism) processors, and propose new techniques to tolerate the late detection of memory traps. With these techniques, the performance cost of a trap is much less sensitive to the location where the trap is detected, even deep in the memory system.

This dissertation demonstrates that virtually-addressed memory hierarchies are feasible and efficient, that the hardware to support them scales much better than for physical memory hierarchies, and that they open new opportunities for future computer architectures.

KEYWORDS: Virtual Memory, Distributed Shared Memory, TLB, Trap, NUMA, COMA, Processor Microarchitecture, Memory Consistency, Performance Evaluation

Chapter 1

INTRODUCTION

Microprocessor performance is continually improving at a fast pace by increasing clock rate and instruction level parallelism (ILP). The memory hierarchy of the processors must satisfy multiple memory accesses in every cycle at a rate currently approaching 1GHz. It must sustain high speed and high bandwidth memory accesses in the face of the growing disparity between processor execution rate and main memory access time, and of the growing memory demand of emerging applications. One additional complication is that, in the process of accessing memory, the virtual address issued by the processor must at some point be translated into a physical address to support virtual memory. Currently, this dynamic translation between virtual and physical addresses is supported by a translation lookaside buffer (TLB). Typically the virtual address translation is performed before or in parallel with the first level cache access, so that the memory hierarchy is accessed with physical addresses throughout[92][83]. We call this current design of memory system physically-addressed memory hierarchy.

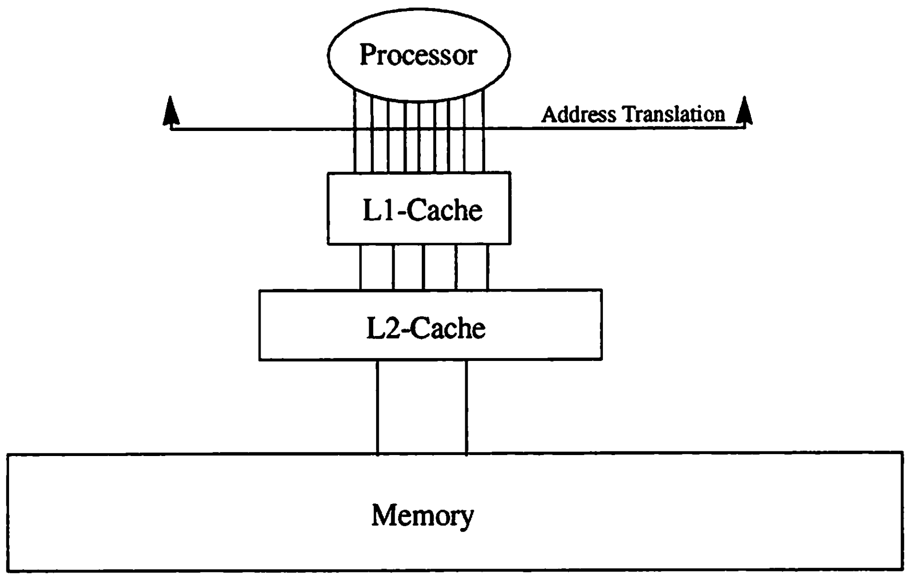


Figure 1.1 A Bandwidth Hierarchy

In this dissertation, we advocate virtually-addressed memory hierarchies where the TLB is removed from the processor and located within the memory hierarchy. Virtually-addressed memory hierarchies cut the overhead and improve the scalability of virtual memory implementations. Moreover, virtually-addressed memory hierarchies open new opportunities to build more flexible and smarter memory systems.

Attacking the TLB Bottleneck. The TLB of a physically-addressed memory hierarchy is a hardware bottleneck, because it must be accessed before or in parallel with time-critical accesses to the first level cache integrated very closely with the processor core where the chip real-estate is very precious. To satisfy the access constraints of the first level cache, the latency and bandwidth requirements of the TLB must scale up with the clock rate and instruction level parallelism[3]. It is getting more difficult and costly to implement a large TLB meeting these speed and bandwidth requirements. Figure 1.1 illustrates a bandwidth hierarchy for a typical computer system, where the dynamic address translation has to meet the peak bandwidth requirement.

The TLB inside the processor core does not scale with the growing working set size of applications and with the size of physical memory. For a given application, the miss ratio of the TLB is primarily determined by the TLB reach, or coverage[19]. New applications are emerging, and at the same time the working sets of applications keep growing and changing. The TLB is fixed within the processor and is very difficult to scale with the applications. It can not be changed when building various computer systems using the microprocessor chips.

Shared memory multiprocessors have become commonplace recently. Small and medium scale multiprocessors are already very successful commercially. Large scale multiprocessors have also been built and will become more and more popular in the near future. The current design of TLBs scales very poorly with the number of processors in a multiprocessor. Multiprocessors tend to have larger memory size and run larger applications. In a multiprocessor, some TLB entries are replicated, wasting TLB space but, more importantly, creating a consistency problem[81]. Maintaining TLB consistency is very expensive and does not scale well. The overhead of maintaining TLB consistency tends to grow in the presence of new optimizations such as page migration for NUMA(Non-Uniform Memory access Architecture) machines to reduce the number of remote data accesses.

Translation misses in the TLB trigger expensive "table walks" through levels of page tables located in main memory. It has been shown in the past[22][54] that the execution overhead due to TLB handling was about 5% - 10% of the total execution time. However, current technology and application trends put more pressure on the dynamic address translation hardware. Some recent studies have shown that the TLB service time alone can consume up to 50% of the user execution time in some workloads[65][79].

In virtually-addressed memory hierarchies where the cache is virtually indexed and tagged, most memory accesses are completed without TLB involvement. The actual address translation, performed only when it is needed, can be done in different locations in the memory hierarchy, and can be implemented in various ways. Putting the TLB after the virtual address cache hierarchy can dramatically reduce the number of TLB misses because of the filtering effect of the caches and also because the TLB may be much larger given the much reduced speed and bandwidth requirements. When the TLBs are shared at the main memory in multiprocessors, the number of TLB misses may become insignificant because of sharing and prefetching effects. The size of the TLB is no longer fixed within the processor chip and therefore can scale with the main memory size. Moreover, maintaining TLB consistency can be eliminated, which largely improves the scalability of virtual memory systems for multiprocessors.

Supporting Generic Memory Function Paradigm. Virtual memory is actually nothing more than a special memory function which provides each user program with the illusion of an exclusive memory space hiding the complexity of managing the physical memory. In general, a memory function is the method or action taken to maintain a particular semantics of the memory system. As a generic programming paradigm, memory functions can be extracted from application programs so that the programs are split into separate parts. The part of memory functions maintain a memory system with some particular semantics or properties which are assumed by the rest parts of the programs. The semantics or properties are usually well-defined to improve the performance, portability, and programmability. For example, the abstraction of virtual memory improves program portability and programmability. As another example, we can consider memory forwarding[49] which maintains the correctness of dynamic memory relocation, so that aggressive run-time data layout optimization is enabled to enhance spacial locality, facilitate prefetching, and avoid cache conflicts and false sharing.

The memory functions express a level of parallelism that can be exploited in coarse granularity. Unlike the SPMD (Single Program Multiple Data) model where all the programs and processing units are somewhat symmetric, memory functions have different functionality and of course quite different behaviors with the original user application programs. It has big potential to take advantage of techniques such as processing-in-memory (PIM) and simultaneous multithreading(SMT).

Traditionally, the support for memory functions has been limited to the support for virtual memory implementation. The TLB handles the most common cases in hardware by caching the recent translation entries, and a trapping mechanism is provided so that uncommon cases such as TLB misses and page faults are completed in software. Recent examples of memory functions such as the concurrent garbage collection algorithm[2] and the original shared virtual memory implementation proposed by Kai Li[48] rely on the virtual memory support.

A virtually-addressed memory hierarchy provides more flexible support for memory functions than the traditional physically-addressed memory hierarchy. This is because of two reasons: 1) In order to support virtually-addressed memory hierarchies, the processors have to be able to take traps from the memory hierarchy. Therefore, special hardware devices can be inserted in the memory hierarchy to handle common cases for particular memory functions, and to generate traps to trigger memory function software for complicated cases. 2) Because the inserted hardware devices can "see" virtual addresses, the memory system semantics can be defined in the virtual address space, and the memory function implementation can be done at the user level. As an example, consider again memory forwarding. First, trapping from main memory is required in case a memory access hit on an indirect relocation pointer. Second, if virtual address is available, the trap handler can be dispatched on any processor (a PIM processor close to the data is preferred but not required) with the correct user context, running in parallel with the computing program.

Fighting the Memory Wall. Microprocessor performance has improved at a rate of 60% per year since the mid 80's, while in contrast the DRAM access time has lagged behind, improving at only 7% per year. Although many latency tolerance techniques such as prefetching and memory consistency models have been exploited to mitigate the gap between processor and memory, it has been shown that the memory bandwidth will soon become a major bottleneck[6], especially given that most latency tolerance techniques consume extra memory bandwidth.

A radical solution to attack this so-called "memory wall" problem is processing-in-memory (PIM), which is enabled by recent VLSI technology integrating processing logic with DRAM memory chips. PIM processors can exploit the huge internal bandwidth associated with the memory banks within the memory chip, as well as the much reduced memory access latency. PIM processors are usually much simpler and may be slower as compared to state-of-the-art microprocessors which exploit parallelism to increase the instruction execution throughput.

Future high performance computer systems should take advantage of in-memory processing, as well as of the increasing on-chip instruction level and thread level parallelism, as illustrated in Figure 6.1. Data-intensive programs such as index searching and virtual memory function run far more efficiently in memory than on ILP processors. Virtually-addressed memory hierarchy is a necessary step to integrate PIM processors into general purpose computing systems. With virtual addressing of memory, system software such as operating system functions can be migrated into memory so that they can executed more efficiently in parallel with user programs, but more importantly, user-level in-memory computing can be naturally supported.

The communication between processor and memory is a major bottleneck because of the big performance gap. For well-behaved applications with good temporal and spatial locality, the cache hierarchy within the processor chip can reduce the necessary communication bandwidth across the processor and memory. PIM processors can execute memory-intensive tasks within memory, which improves the system performance at least in 3 directions: 1) The necessary communication bandwidth between processor and memory is reduced because the processing is absorbed within the memory closer to the data; 2) The memory-intensive tasks can be executed much faster by PIMs than by the ILP processors because higher clock rate or deeper pipelining will not improve their performance; 3) The processors can run compute-intensive applications in parallel with the PIMs, increasing the overall throughput of the system.

Memory functions are a perfect paradigm to split the application programs and distribute the work among processors and PIMs. The memory functions can be executed either in PIMs or on ILP processors. While cache-friendly memory system properties can be defined and maintained to reduce the memory bandwidth, portable and efficient PIM programs can be written to execute various layers of memory functions in memory, eventually leading to a smooth transition path for PIM technology to the general-purpose computing.

1.1 Research Contributions

In this dissertation, we evaluate the performance issues associated with virtually-addressed memory hierarchies and propose new ideas to support and exploit virtually-addressed memory hierarchies.

- Various memory systems are compared where the address translation is done in different places in the context of large scale multiprocessors. Our evaluation results show that because of the filtering of the virtual address caches the number of address translation misses is dramatically reduced when the TLB is moved down to the memory hierarchy.

- Novel multiprocessor architectures are proposed and evaluated. In one of them, called the virtual COMA(V-COMA) architecture, the entire memory hierarchy is accessed with virtual addresses. The dynamic address translation mechanism is shared among all the processors, and is combined with the cache coherence protocols. The translation overhead is reduced to a minimum and the consistency problem is eliminated. V-COMA scales well and even works better with larger number of processors.

- A new solution to the synonym problem is introduced to enable virtually-addressed memory hierarchies. The major idea is to replace the TLB in the processor with a Synonym Lookaside Buffer (SLB), which translates synonyms into unique identifiers to address the memory hierarchy. An SLB can remain very small because its size depends on the sharing of synonyms, not on the size of applications or of the physical memory.

- Using contemporary workloads running on a real operating system (SGI Irix 5.3), we compare the performance of virtually-addressed caches and traditional physical address caches. It appears that virtual address caches have better miss rates than physical caches. In particular, our proposed solution using a small SLB in front of the caches avoids short misses in large caches, while safeguarding the benefits of the temporal and spatial locality presented in the virtual address stream.
- The behavior of traps in modern ILP processors is analyzed in-depth. Several metrics are defined to quantify the overhead of traps. In particular, the performance impact of late memory traps is identified.
- Three techniques are proposed to tolerate late memory traps. The performance cost of any trap is very high in ILP processors. At the same time, using our techniques, the overhead due to the lateness of detecting traps is considerably lower. Therefore, with our techniques, the overhead of traps is not much sensitive to the location where the trap is detected.

1.2 Organization of the Dissertation

In chapter 2, we describe some background material for this research. We start with a brief overview of virtual memory systems, followed by an introduction of the architecture of contemporary microprocessors, which are the building blocks of most computer systems. Then we briefly describe some multiprocessor architectures. Multiprocessors are becoming the prevalent architecture for computers from low end desktops to high end supercomputers and commercial servers.

In chapter 3, we look at various design options for virtually-addressed memory hierarchies in the context of large scale multiprocessor systems. In particular, a novel multiprocessor architecture V-COMA is proposed and evaluated.

In chapter 4, we focus on the synonym problem, which is one of the major technical difficulties faced by virtually-addressed memory hierarchies. We propose and evaluate a new scheme to solve the synonym problem, the synonym lookaside buffer (SLB). We also compare the miss rate behavior of physical and virtual address caches.

In chapter 5, memory traps in ILP processors are analyzed in-depth. Several metrics are defined to capture the overhead of traps. In particular, the impact of late memory traps is identified. Three techniques are proposed to tolerate late memory traps.

In chapter 6, we discuss some implications of virtually-addressed memory hierarchies. Virtually-addressed memory hierarchy empowers more flexible and smarter memory systems enhanced with the general memory functions and processing-in-memory (PIM). In chapter 7, we summarize the related work. In chapter 8, we present the conclusion of this research.

Chapter 2

BACKGROUND

In this chapter, we describe some background material for this research. We start with a brief overview of virtual memory systems, followed by an introduction of ILP processors. We also briefly describe multiprocessor architectures.

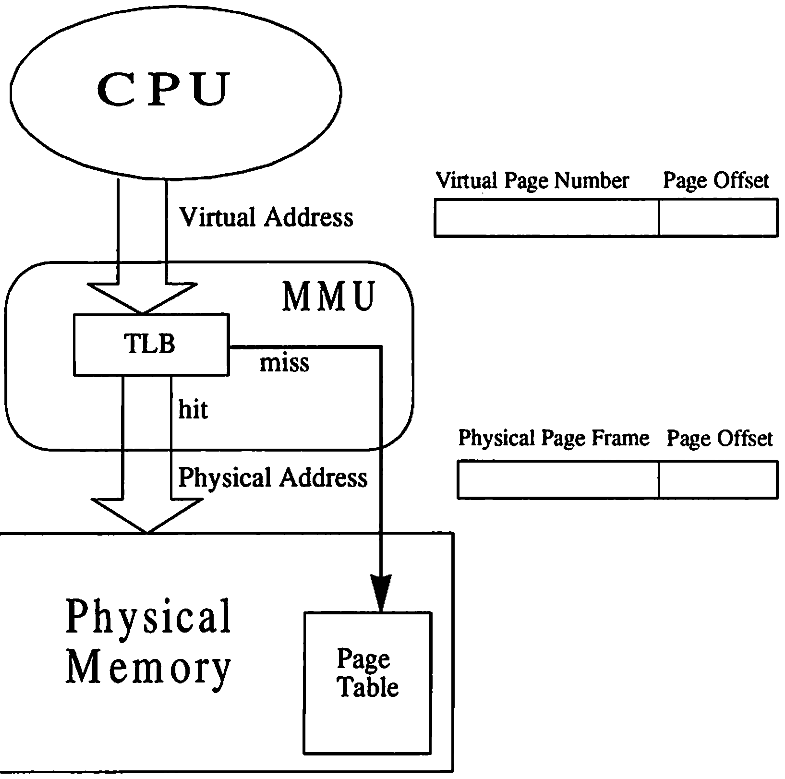


Figure 2.1. Virtual Memory Implementation

2.1 Virtual Memory

Virtual memory system manages physical main memory and provides each individual user program an illusion of a single large memory space. Through cooperative hardware and software support, the virtual memory system automatically moves data between secondary storage and main memory. User programs running in their virtual address spaces are no longer concerned with physical memory management.

Figure 2.1 illustrates a typical implementation of a virtual memory system. Most virtual memory systems implement demand paging, where the physical memory is divided into fixed pages, typically 4K or 8K bytes. Each virtual address is dynamically translated into a physical address to access physical memory. If the accessed data is not present in physical memory, a page fault trap is generated and the system software loads the page from secondary storage, updates the virtual-physical page mapping and resumes the execution of the user program.

Typically, the virtual-physical page mapping is maintained by page tables which are special data structures maintained by the virtual memory system software. There exist different page table organizations. The two major structures are forward page table and inverted page table. Forward page table is indexed with the virtual page number. Because the virtual address space is usually huge and very sparse, a naive one level page table is wasteful of memory. A multi-level page table structure saves memory by taking a hierarchical approach. In multi-level page tables, several memory accesses are required to translate a virtual address. An example of a forward page table structure is shown in Figure 2.2. An inverted page table is indexed by the physical page number instead of the virtual page number. The size of the inverted page table is proportional to the size of main memory. The inverted page table can not be directly accessed by virtual addresses. A hash function is necessary to find the physical page number from the virtual address. An example of inverted page table structure is shown in Figure 2.3.

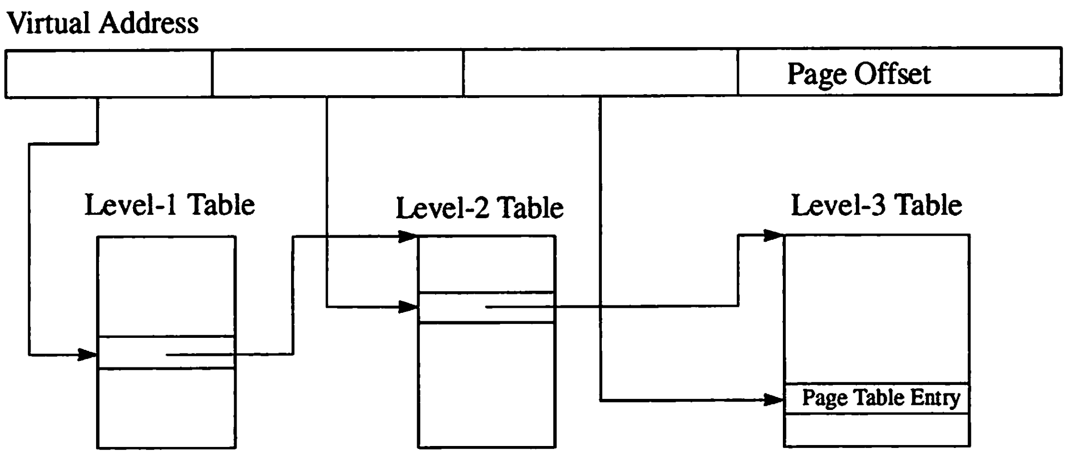


Figure 2.2. Forward Page Table Structure

Logically, the main memory is organized as a fully associated cache controlled by the page table with the block size equal to the page size. Virtual memory system software manages the data movement between main memory and backup secondary storage, typically a disk. Because of the huge latency gap between main memory and disk I/O access, usually software controlled context switches are necessary to tolerate the latency in case of a page fault.

A special cache commonly called TLB (Translation Lookaside Buffer) is used in almost every current computer system to accelerate the dynamic address translation. The TLB caches the most recently used page table entries is typically located very close to the processor pipeline and is accessed before or in parallel with the first level cache. If a virtual address hits in the TLB, it is immediately translated into a physical address by the matching TLB entry. If it misses, the in-memory page table is looked up to complete the translation. If the look up procedure is successful and the page table entry is valid, it is inserted into the TLB and the program execution is resumed, otherwise a page fault is generated. The TLB either is fully associative or has a high degree of associativity in order to reduce the number of conflict misses.

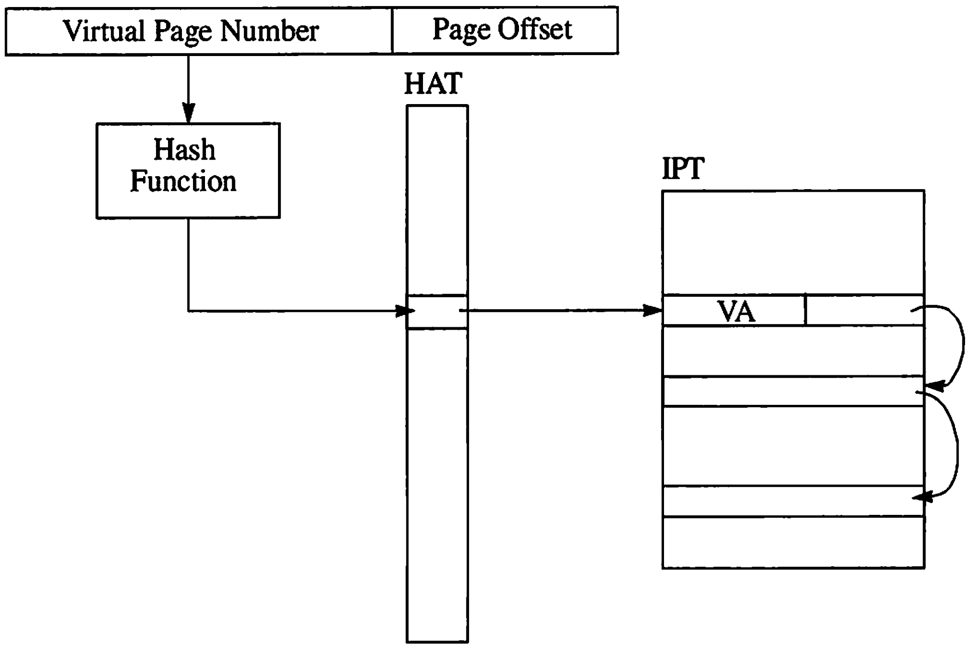


Figure 2.3. Inverted Page Table

Figure 2.4 illustrates a typical entry of the TLB. Besides the virtual and physical page numbers, a hardware context identifier is included to avoid flushing the entire TLB on every context switch. The access right bits restrict accessibility to the page. Typically, a memory trap is generated if access rights are violated. The entry is invalid if the valid bit (V) is reset. The reference (R) and dirty (D) bit are usually needed by the software to optimize the paging algorithm. Some TLBs include a global bit which allows the TLB entry to map a global virtual address area across all the contexts. Many TLBs support the mapping of multiple page sizes in power of two. Such pages are called superpages and their purpose is to increase the coverage of the TLB mapping.

context	virtual page number	physical page frame	access right	R	D	V

Figure 2.4. TLB Entry

This is a very brief summary of the basic concepts in a virtual memory system. Any real implementation is far more complex. Memory management is now the heart of almost every operating system.

2.2 ILP (Instruction Level Parallelism) Processor

With the advance of VLSI technology, the architecture of computer systems is centered around microprocessors[24]. Almost all modern computers from laptops to supercomputers are built with microprocessors because of technological and economic reasons. In order to achieve high performance, modern microprocessors use very complex microarchitectures to exploit instruction level parallelism(ILP).

Figure 2.5 shows the block diagram of a typical ILP processor similar to the MIPS R10000[92]. Multiple instructions are fetched and decoded in program order, execute out-of-order in the execution engines, and finally graduate in program order again.

In the instruction fetch stage, the addresses of instructions following the branch are predicted. In the decode stage, logical registers are renamed to physical registers, and the decoded instruction is appended to an active list and put into one of the instruction queues waiting to issue. When a branch instruction is decoded, the processor state is saved in a branch stack, from which the state can be recovered in case the branch is mispredicted. A branch mask associated with each instruction in the instruction queues and in the execution pipelines points to the depending branches. If any one of these branches is mispredicted, the instruction is aborted. The instruction queues act as reservation stations, monitor the register write ports and issue the instructions to the pipelined execution units based on data flow dependencies. When an execution unit completes an instruction, it sends the instruction tag to the active list. Instructions are retired in program order after they are completed. When retiring an instruction, the active list commits the register mapping and returns the physical registers to the free list.

Retiring instructions in order is critical for handling precise interrupts[75]. An exception can only be taken when the instruction reaches the top of the active list, at which point all previous instructions have successfully retired. The pipeline is flushed and the state is recovered by unmapping the registers in reverse order of instructions in the active list, usually at the same rate as the decode rate.

2.3 Shared Memory Multiprocessors

Shared memory multiprocessors are the prevalent architecture to achieve performance levels beyond that of single processors. Most supercomputers and high end servers are multiprocessors, and it is now very common to see multiple processors in low end servers and desktops. Considering technology and architecture trends, single chip multiprocessor products are likely to appear in the near future.

2.3.1 Architecture

Most small scale multiprocessors are bus-based symmetric multiprocessors (SMP). As shown in Figure 2.6, all the processor modules as well as the main memory and I/O devices are attached on the snooping bus. Cache coherence is maintained by the snooping protocol. Memory accesses that miss in the processor cache are broadcast on the bus, and are snooped by all the

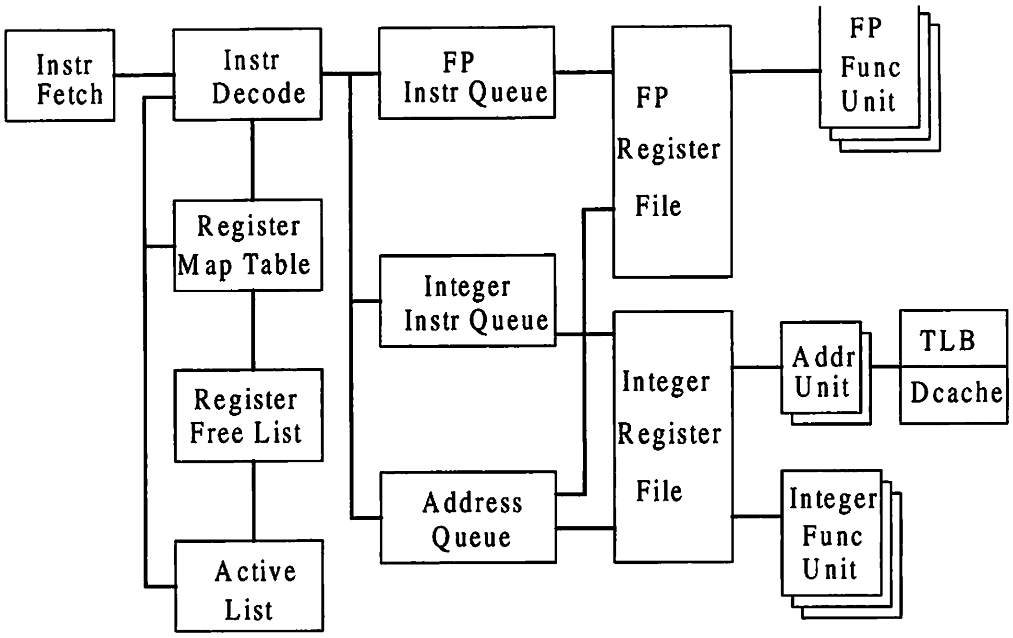


Figure 2.5. Block Diagram of an ILP Processor

caches. Depending on the coherence protocol, either another cache or the main memory supplies the data to complete the memory access.

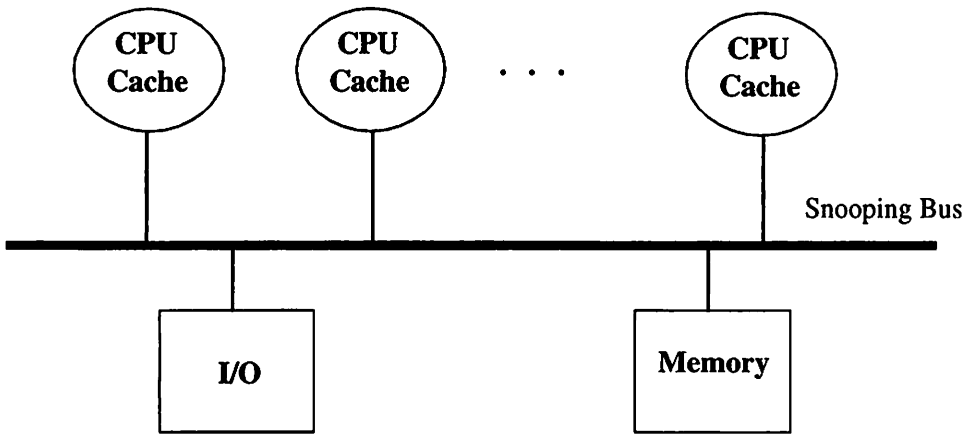


Figure 2.6. Bus-Based Multiprocessor

Bus-based multiprocessors do not scale well because they are restricted by the bandwidth of the snooping bus. For large scale multiprocessor systems with hundreds or even thousands of processors, directory-based distributed shared memory systems become the natural design choice. Each node in a distributed shared memory system contains a portion of its main memory which is shared and can be addressed by all the processors. Because the system is not limited by a centralized broadcasting mechanism such as the snooping bus, the distributed shared memory system can scale up to a very large number of processors. Usually, a directory-based cache coherence protocol is used, where the directories contain the copy information for each memory block.

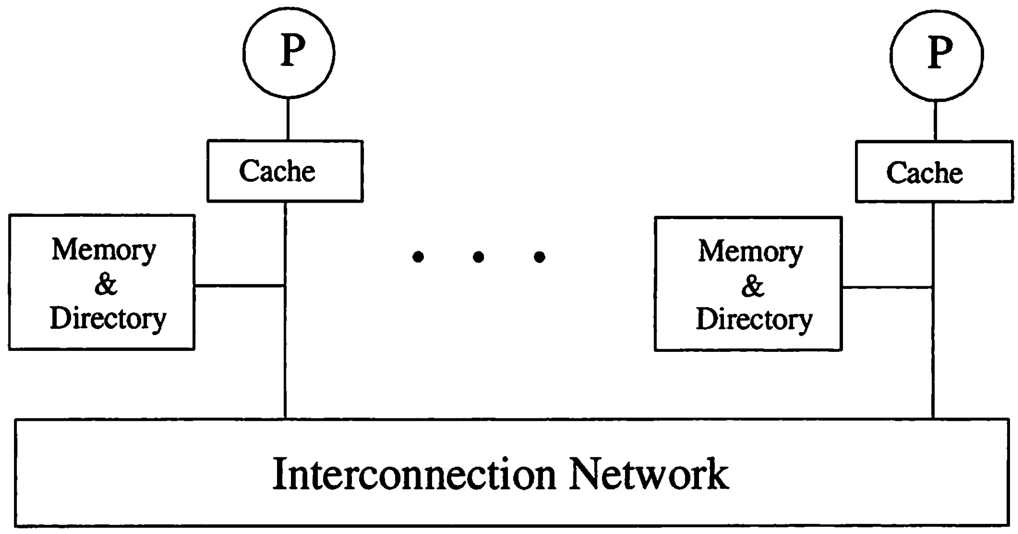


Figure 2.7. CC-NUMA Architecture

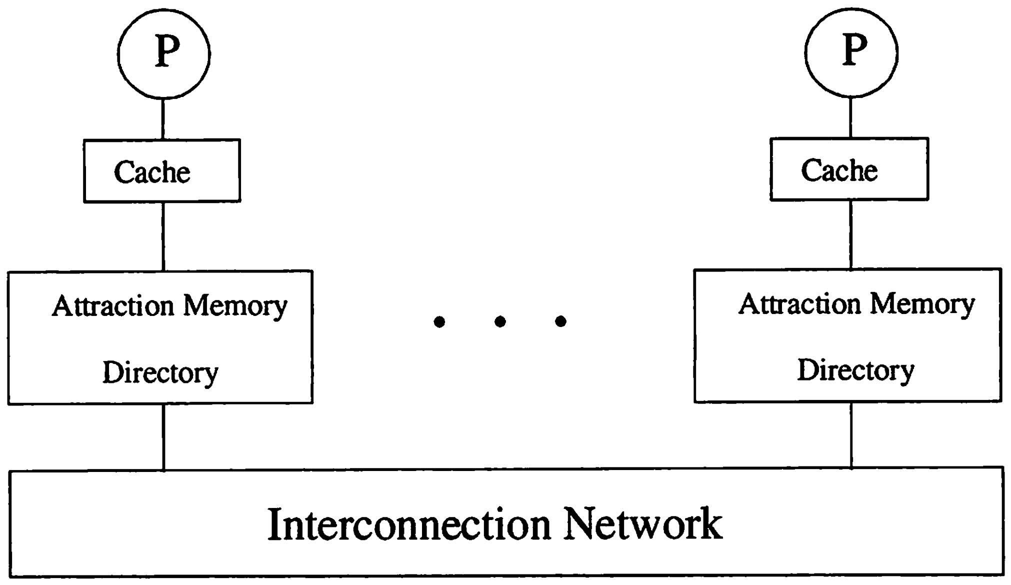


Figure 2.8. COMA Architecture (COMA-F)

Cache Coherent Non-Uniform Memory access Architecture (CC-NUMA) and Cache Only

Memory Architecture (COMA) are two major variations of the large scale distributed shared memory systems. Figure 2.7 shows a typical CC-NUMA architecture. In CC-NUMA, the data replication and migration happen in the processor caches. If a memory access misses in the cache, a memory request has to be sent to the home node of the memory block, which could be either remote or local to the processor. Based on the directory at the home node, appropriate coherence actions will be taken to complete the memory request.

Figure 2.8 shows a typical COMA architecture. Here we consider a flat COMA[44] instead of a hierarchical COMA[35] architecture. In COMA, the main memory is organized as another level of cache which is called attraction memory. Automatic data replication and migration can happen in the attraction memory. After a memory access misses in the processor cache, the local attraction memory is first looked up. If the data is not present in the attraction memory, the request has to be sent to the home node where the directory information resides indicating the location of the copies. The home node does not necessarily contain the data as in CC-NUMA. The COMA protocols are much more complex to implement compared to the CC-NUMA protocols.

2.3.2 Memory Consistency Model

In order to guarantee the correct execution of parallel programs, the memory consistency model must be respected[25].

A memory consistency model specifies the constraints on the order in which memory operations must be globally performed (i.e. become visible to all the processors)[25]. The most intuitive and natural model for programmers is the sequential consistency(SC) model, which is defined by Lamport[50] as the following:

A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Sequential consistency enforces very strict constraint on orders of memory accesses, which restricts many performance optimizations that modern uniprocessor compilers and microprocessors employ. There are other different memory consistency models that relax the ordering restrictions of sequential consistency. Total store order(TSO) and processor consistency(PC) relax the write-to-read order. Partial store order (PSO) relax both write-to-read and write-to-write orders. Weak ordering(WO) and release consistency (RC) relax all memory access orders and rely on synchronization instructions to guarantee correct execution of programs.

2.4 Evaluation Methodologies

In order to understand the issues and evaluate various ideas, we simulate the architectures using trace-driven and execution-driven simulations.

Trace-driven simulation is used to study the performance for memory hierarchy in the uniprocessor environment. The trace is a sequence of load/store memory accesses from the application workload, which drives the cache/memory simulator to evaluate the performance. Trace-driven simulation is simple and fast. However, it has many limitations. Only the behavior of memory system is accurately evaluated, the overall execution time and other system components including the processor are not simulated. In addition, it is not accurate for multiprocessor evaluation.

We also use execution-driven simulation in this research. Typically, an execution-driven simulator models the function and timing for all the components of the target computer system, including the processors, the cache/memory system, bus, interconnection network, and I/O. The models can have different levels of detail depending on the accuracy and simulation time trade-off. The application executables are run on top of the simulator just as they run on the target machine. The simulated processor interprets and executes the application instructions based on the appropriate timing. Usually, there will be an event engine that maintains an internal global time and schedule the order of events.

Chapter 3

VIRTUALLY ADDRESSED MEMORY HIERARCHIES

In this chapter, we look at various design options for virtually-addressed memory hierarchies in the context of large scale multiprocessors. The basic idea is to move the address translation closer to memory, where the TLBs are shared, do not have coherence problems, and scale well with both the memory size and the number of processors.

3.1 Virtual Address Cache Issues

Virtual address caches can relieve the latency and bandwidth requirements of the TLBs. When the cache is virtually indexed and tagged, most memory accesses are completed without TLB involvement; in fact, address translation can be done in different locations in the memory hierarchy[85], and can be implemented in various ways, such as in-cache translation[89] or even by software[42].

There are a series of issues related to any system with virtually indexed and virtually tagged caches, such as the synonym problem for which we will propose a new solution in chapter 4. We briefly summarize these issues here.

3.1.1 Synonyms and Address Mapping Changes

Virtual address caches suffer from synonym and address mapping change problems. Synonyms happen when multiple virtual addresses map to the same physical address and may cause inconsistencies in a virtual address cache. Virtual-physical address mapping changes are due to deallocation and reallocation of pageframes. Since mappings may still remain in the virtual address cache after a page has been demapped, they must be flushed to avoid inconsistencies.

Chapter 4 analyzes the synonym problem in detail. In this chapter, we assume that the virtual addresses that are used to address the memory hierarchy do not contain synonyms. A PowerPC-like segmented memory system or a single address space operating system can create a global virtual address space where synonyms are not allowed. The SLB scheme proposed in chapter 4 can also be used to generate unique virtual addresses for synonyms.

3.1.2 Write-backs and Inclusion

Besides cache misses, stores must also propagate down the hierarchy as write-through or write-back accesses. Because write-backs may not be part of the current working set of the applications, they have a higher probability of missing in the TLB which is located right after the virtual address caches.

Usually, inclusion is maintained between the caches. Thus, when a block is removed from a lower level cache, the caches up the hierarchy must be invalidated. When the TLB is inserted in the cache hierarchy such as the L1-TLB shown in Figure 3.1, the caches above the TLB are indexed in virtual addresses, while the caches after the TLB are indexed in physical addresses. A reverse translation mechanism, usually in the form of backpointers, is needed between a physical address cache and the virtual address cache above it[85] in order to maintain the inclusion property between the caches.

Maintaining inclusion between the TLB and the virtual address cache memory above it avoids TLB misses on write-backs from the cache. Although the TLB placed after a virtual address cache can be very large and slow in uniprocessors, inclusion is expensive in multiprocessors. First, whereas a large cache cuts the number of capacity misses dramatically, coherence misses can not be filtered out and the longer address translation latency impacts coherence operations. Second, the TLB size to maintain inclusion grows with the cache size, leading to higher cost and longer latency. Third, it is not mandatory to maintain inclusion. Physical pointers stored in the virtual address cache can avoid accesses to the TLB on a write-back, just as pointers are used in the physical address cache to access the virtual address cache above it[85].

Because of the effect on coherence misses and the sheer cost of fast, large TLBs, we do not enforce inclusion between a TLB and the virtual address caches above it.

3.1.3 Late Detection of Memory Faults

After a virtual address is translated into a physical address, the processor assumes the memory access will complete without fault (except for fatal errors) because the TLB only contains mapping for valid pages in memory. The traditional TLB coverage thus provides a "safe-access subset" such that any memory access that passes the TLB does not generate any exception in normal execution. However, this is a conservative strategy, in which every memory access outside the subset stalls the processor even though it may not lead to a memory fault.

Putting the address translation mechanism after the virtual address cache postpones the decision time of the memory fault. The virtual address cache and the following TLB expand the "safe-access subset", which means that a page fault or a miss of the TLB will trap the processor after a longer detection time as compared to a system with physical caches. This issue of late detection of memory traps is studied in chapter 6, where the trap in ILP processors is analyzed and new techniques are proposed to tolerate the late memory traps.

3.1.4 Access Rights

Each page table entry contains protection bits such as read, write, and execute for the page. These attributes migrate to the TLB on a TLB miss. Thus, in a system with virtual address caches access right bits must be copied at least in the first level cache. This requires cache flushing when access rights are changed. Of course in a segmented system as the PowerPC architecture,

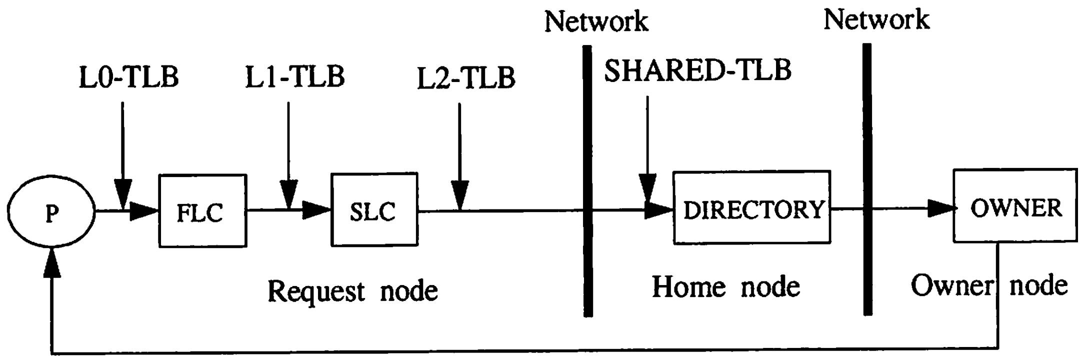


Figure 3.1. Possible Locations for the TLB in CC-NUMA

access rights can be easily checked at the segment granularity[42][52] by storing access right bits in the segment register.

3.2 Options for Dynamic Address Translation

3.2.1 Dynamic Address Translation in CC-NUMA

Address translation can be done at different levels of the memory hierarchy in a traditional CC-NUMA architecture, as shown in Figure 3.1. Most processors translate virtual addresses in a TLB before or in parallel with the first-level cache (L0-TLB). However, provided caches are virtually indexed and tagged, the TLB could be placed between the first- and second-level caches (L1-TLB) or after the second-level cache (L2-TLB). In these cases, the TLBs are private and their consistency must be maintained. Alternatively, the TLB could be associated with the home node (SHARED-TLB). In this latter case, the TLBs are shared, map the local memory only, and do not cause coherence problems. However, because the home node is selected with the virtual address the programmer has no control over page location and page migration is impossible. Because page placement cannot be optimized for locality, capacity misses are remote most of the time resulting in poor performance for applications whose significant working set does not fit in the second-level cache.

3.2.2 Dynamic Address Translation in COMA

The design options for dynamic address translation and their trade-offs in COMA[44] are different than in CC-NUMA. Locating the TLB at the memory in a COMA does not affect the latency of capacity misses much because of the automatic migration and replication of memory lines. We compare five schemes called L0-TLB, L1-TLB, L2-TLB, L3-TLB and V-COMA. These schemes are illustrated in Figure 3.2.

The **L0-TLB** scheme is the traditional dynamic address translation design in most current processors[83][92] and the habitual scheme for a physical COMA. Every memory address issued by the processor is translated by the TLB. All caches and the attraction memory are physically addressed.

The **L1-TLB** scheme puts the address translation mechanism after a virtual FLC (first-level cache), but before a physical SLC (second-level cache) [85]. Backpointers in the physical address SLC are needed to maintain inclusion. The L0-TLB and L1-TLB schemes are not very different for CC-NUMA and COMA.

In the **L2-TLB** scheme both FLC and SLC are virtual address caches. The software-managed address translation scheme proposed in [42] can be seen as an L2-TLB scheme which has 0 entry and traps the processor on every SLC miss. The L2-TLB scheme for COMA is different than the L2-TLB scheme for CC-NUMA in that backpointers are needed in the attraction memory to maintain inclusion.

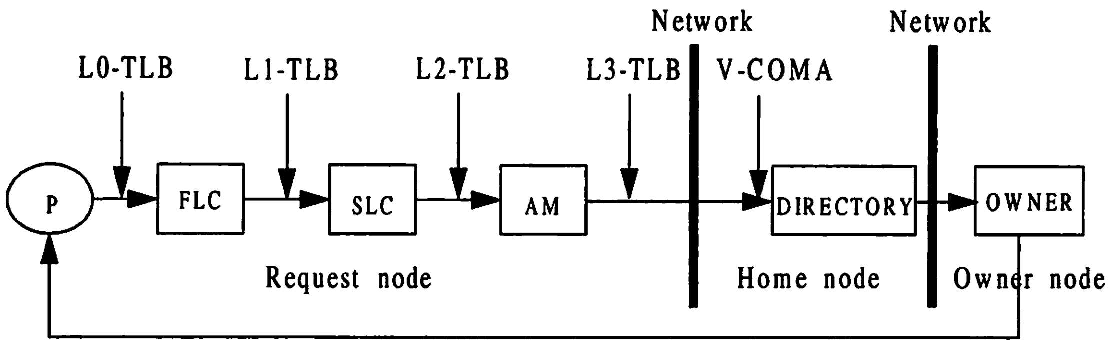


Figure 3.2. Possible Locations of the TLB in COMA

In a COMA the attraction memory acts as a cache and may also be virtually indexed and virtually tagged. In the case of **L3-TLB**, address translation is postponed until a miss in the local node, as shown in Figure 3.2. The coherence protocol is maintained with the physical address, which points to the home node.

In a typical COMA, the attraction memory in every node is divided into an equal number of sets. A *global set* is made of all the sets with the same number in all attraction memories, as is illustrated in Figure 3.3 where each attraction memory is 4-way set associative. The size of a *global set* increases linearly with both the number of processing nodes and the set size in each attraction memory.

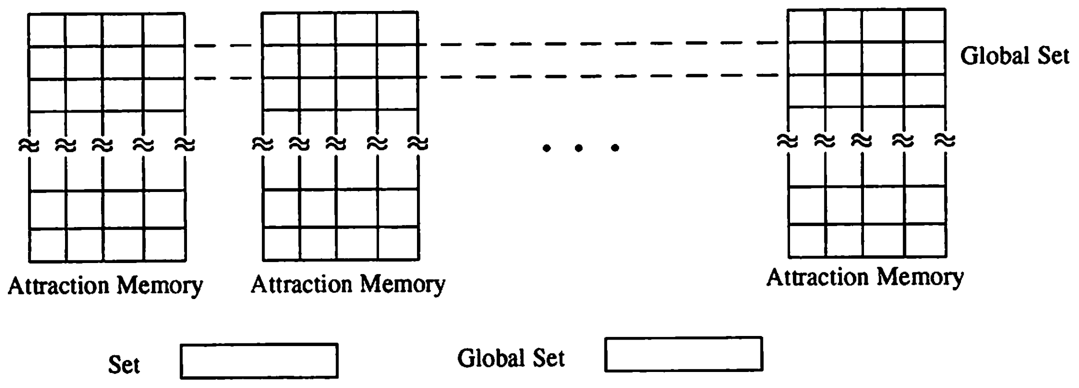


Figure 3.3. Set and Global Set

For a physically indexed attraction memory a virtual address can be mapped into different global sets depending on the virtual-to-physical address mapping. By contrast, in the L3-TLB scheme, a data block is restricted to reside in the global set indexed by its virtual address. The number of *slots* for a given block is limited by the size of a global set. A page occupies the same slots in consecutive global sets, so that we can also speak of the *slot of a page*. The *global page set* is made of all the contiguous global sets in which the blocks of the page can reside. The number of slots for a page is limited by *memory pressure*, which is given by the number of slots occupied in a global set divided by the size of the global set. When the pressure approaches 1, replication in the global set is inhibited. Therefore, if the pressure in the global page set to which a new page maps is too high, a page must be swapped out even though the pressure in other global page sets is low. In a nutshell, the virtual-to-physical address mappings are set-associative instead of fully associative and the set size is equal to the size of a global page set.

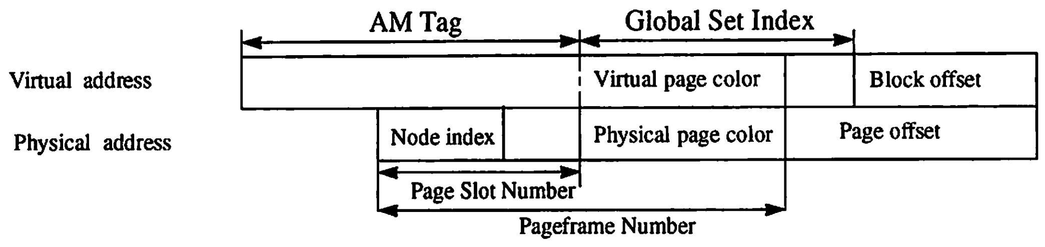


Figure 3.4. Page Coloring in the Attraction Memory for L3-TLB

This page allocation strategy is equivalent to page coloring applied to the attraction memory. Page coloring allocates virtual pages to physical pageframes sharing the same least significant bits so that the virtual address and the physical address index to the same set of the cache[51]. As shown in Figure 3.4, if the virtual and the physical addresses have the same color, the set index for the data in the physically indexed attraction memory is determined to be the same as in the virtually indexed attraction memory. The most significant bits of the physical page number contain the slot number. The maximum number of slots of each *global set* is given by the number of processors multiplied by the associativity of each attraction memory.

3.3 Virtual COMA (V-COMA)

In L3-TLB a TLB is still private to a processor node, which means replication of TLB entries across nodes and TLB consistency enforcement. We propose to move the address translation into the home node and to integrate it with the cache coherence protocol as shown in Figure 3.2. In this new design, which we call V-COMA, the support for address translation is located at the home node.

In V-COMA the attraction memory is accessed as in L3-TLB, but the directory at the home node is accessed with virtual addresses instead of physical addresses. Thus, as in SHARED-TLB, the home node is fixed by the virtual address. Because its architecture is not conventional, V-COMA is described further in the next section.

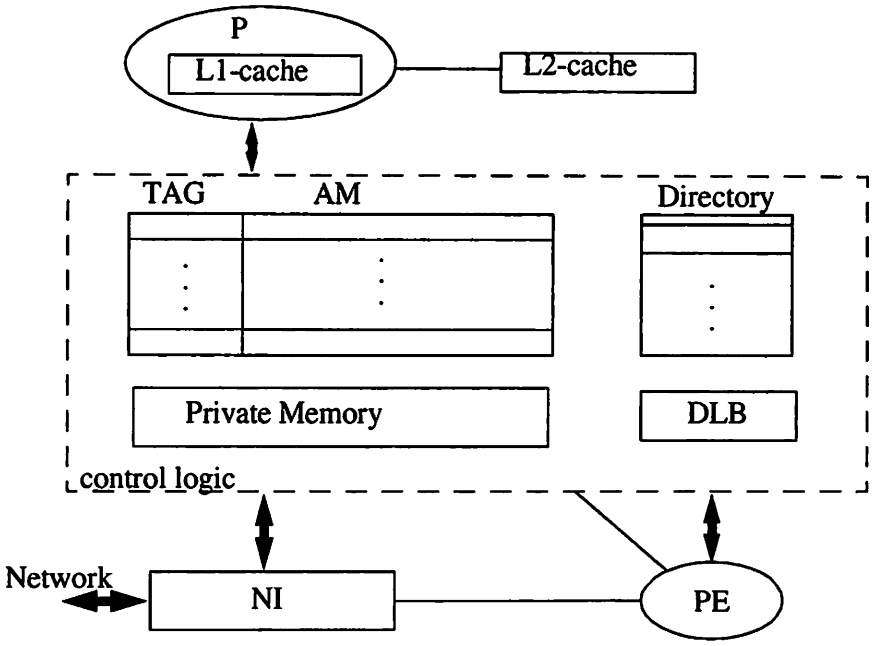


Figure 3.5. Processor Node Architecture in V-COMA

3.3.1 V-COMA Processor Node Architecture

Figure 3.5 shows the architecture of one processing node in V-COMA. The processor P does not need a TLB. Its caches and attraction memory are all virtually indexed and tagged. Some private memory in each processing node stores page tables for local pages and can also be used for private (non-replicated) data. The directory for local pages is stored in private memory or in a separate memory. The DLB (*Directory Lookaside Buffer*) is a cache to speed up translations from virtual addresses to directory addresses in the directory address space. The PE (Protocol Engine) executes the cache coherence protocol and refills DLB entries on DLB misses and is similar to the MAGIC chip in FLASH[47]. PE programs reside in local private memory. If the PE has enough processing ability and bandwidth, it could perform additional functions, such as periodically resetting the reference bits in page table entries.

3.3.2 Directory Organization and Coherence Protocol

Virtual addresses are not suitable to address the directories. Compared to the physical address space, the virtual address space is huge and sparse. Every virtual address issued by the processor is not guaranteed to reside in primary storage, or even exist. The size of the necessary directory memory is determined by the size of the main memory. Since only a small part of the virtual address space resides in physical memory, it is impossible and unnecessary to reserve a directory entry for every virtual address block. Virtual addresses have a limited life time, which makes it more difficult to manage the directory space by virtual addresses. Thus V-COMA moves the virtual address translation into the directory look up procedure by translating virtual addresses into *directory addresses* to find a directory entry.

The directory memory is organized in *directory pages*. Contiguous directory entries in a *directory page* correspond to contiguous memory blocks of a memory page. Therefore, a *directory page* has as many entries as there are blocks in a memory page. The directory memory is allocated and reclaimed in *directory page* unit by the virtual memory system. Due to the set-associative nature of the attraction memory, the mapping of a virtual page to a *directory page* in the page table is also set-associative, where the set is the *global set*.

Figure 3.6 shows how the virtual address is decomposed into fields to access the directory. Assume that the number of attraction memory sets per node is $S = 2^s$, the associativity is $K = 2^k$ blocks per set, the block size is $B = 2^b$ bytes, the number of processor nodes is $P = 2^p$, and the page size is $N = 2^n$ bytes. The p least significant bits of the page number point to the home node and the n-b most significant bits of the page displacement point to the entry within the directory page. s-p-n+b bits of the page number are used to index the *global set* in the page table (of size PxK) where the base address of the directory page can be found. The virtual address tag is then matched to the tags in the set. Since the global set is very large, we must use hashing or hierarchical translation based on the virtual tag to access the page table entry. A detailed page table structure is described in section 4.3.

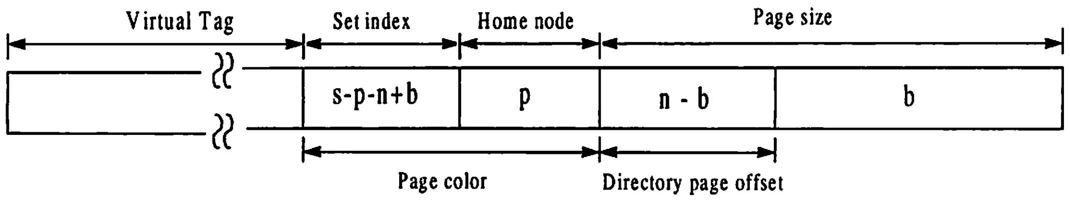


Figure 3.6. Access to the Directory of V-COMA

Because the latency to locate the directory for a memory block is in the critical path of the cache coherence protocol, a DLB is put between the protocol engine and the directory memory in every processing node to accelerate the directory look up procedure. Accesses to the DLB are fully or set associative, as illustrated in Figure 3.7.

The protocol is write invalidate and is basically the same as in COMA-F[44]. Each block in attraction memory can be in one of four stable states: Shared, Master-shared, Exclusive and Invalid. The replacement policy is random. Replacements of Exclusive or Master-shared copies send the block to the home node, which accepts the injection only if it has spare Invalid blocks in the same set. If not, the home node forwards the block copy to a random node. The selected node accepts the injection if it has an Invalid or Shared block available. If not, it forwards the request to the next node.

Cache coherence is maintained using virtual addresses. If a processor access is satisfied at the local node, no address translation is needed. If the access misses in the local node, a request is sent to the home node specified in the virtual address. The home node translates the virtual address to a directory address to find the state and copy set of the accessed block. The rest of the protocol transaction is handled with virtual addresses.

3.3.3 Page Table Structure

Since the virtual address space is set associative, the page table and directory memory are also set associative in the same structure. Each set is independent and detects memory pressure separately. The management of the physical memory can be much easier because of the slot concept instead of a contiguous physical address space and also because of the reduced memory size per global page set. However, a page table structure is still necessary because of the sparse and huge space represented by the virtual tag for each global page set. We describe an inverted page table structure in this section. Other alternatives inspired from conventional page table structures are also possible.

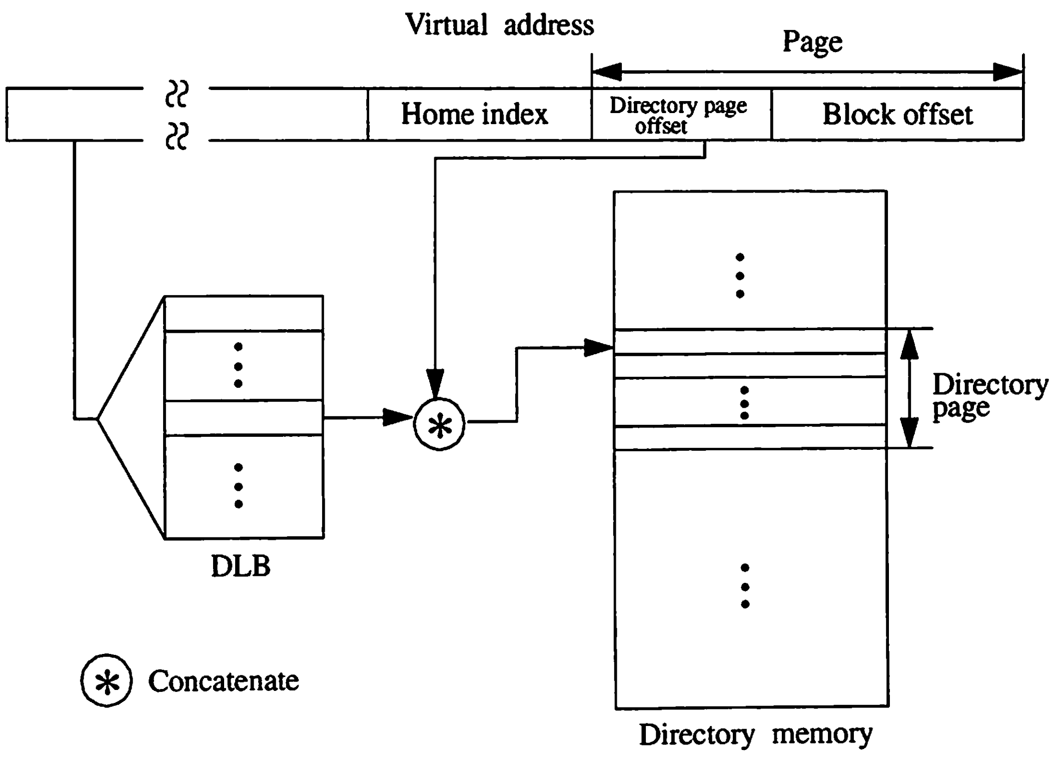


Figure 3.7. V-COMA Address Translation with DLB

Figure 3.8 shows the structure of the page table in V-COMA. It is a set associative inverted page table (IPT). As an IPT, every page table entry connects to a directory page permanently. Hash Anchor Table(HAT), IPT and directory memory are all set associative just as the virtual address space. The home node index bits of the virtual page do not hash, while the other bits in the virtual page number participate in the hashing. The set index bits at each node is the same set index for HAT, IPT and directory memory. The virtual tag which is the most significant part of the virtual address is hashed into the HAT. Hash function is the same but independent for different sets. The base address of the page table and HAT can be stored in the private memory or in a con-

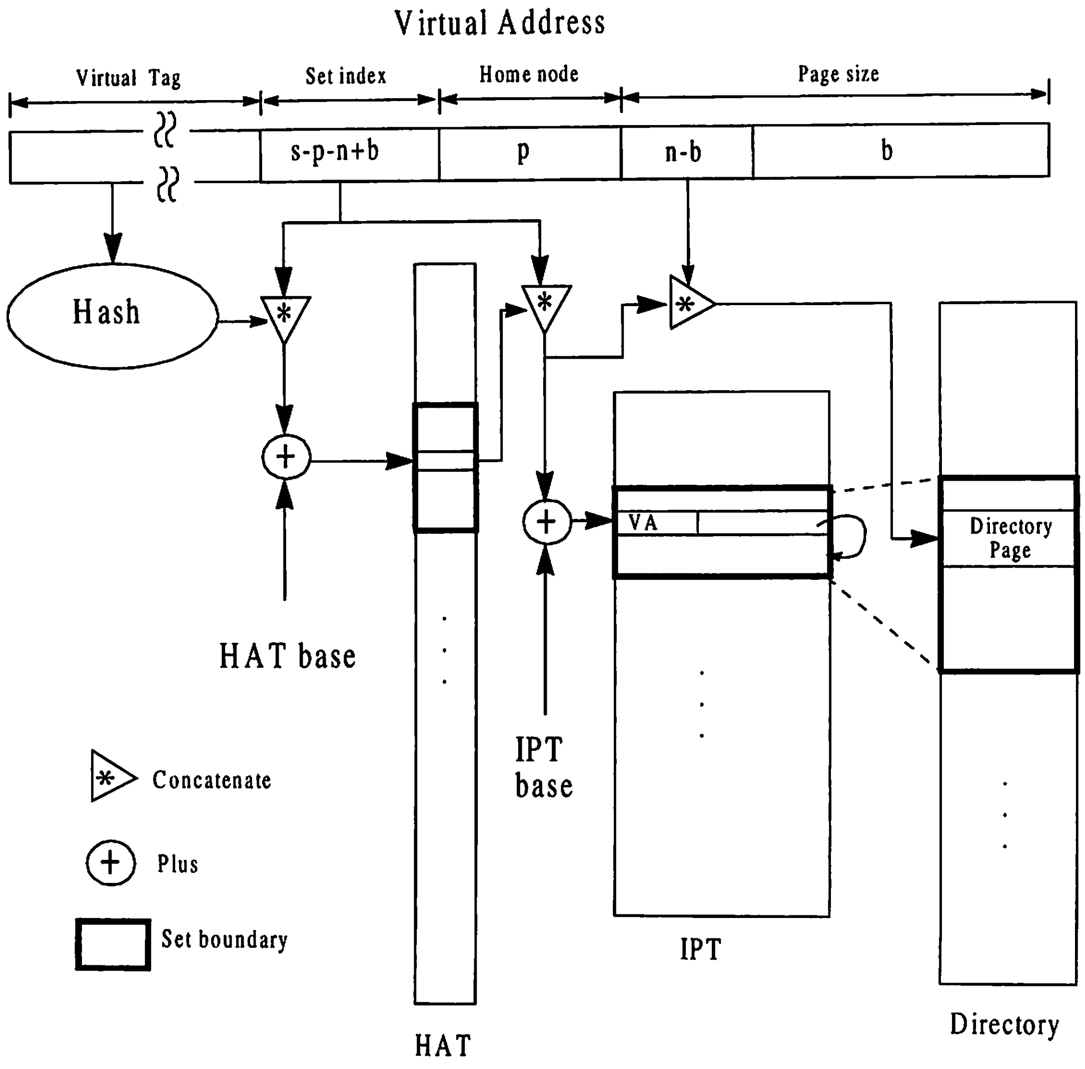


Figure 3.8. Page Table and Directory Memory Organization

trol register if implemented in hardware. The virtual address bits are compared with the tag in the inverted page table entry. If they match, it is the right page table entry and the DLB entry is refilled; otherwise, the next entry is searched through the next pointer in the page table entry. If there is no match at the end of the chain, the virtual page is not resident in the physical memory.

Unlike a forward-mapped page table, there is only one page table entry for each page slot. The size of the page table depends on the size of the primary memory. Therefore, the size of the necessary private memory for the page table scales with the attraction memory size. The page table resides in the private memory on each home node, and guarantees directory accesses be completed locally at the home node.

3.3.4 Impact on Virtual Memory Management

The impact of V-COMA's memory organization on memory management is no greater than for traditional COMAs. From the point of view of virtual memory management, the *directory page* corresponds to the physical pageframe in a classical system.

On a page fault, a *directory page* and a page table entry are requested from the page's home node. A resident page may have to be swapped out by the page daemon if the memory pressure of the page's *global page set* is higher than a threshold. A page table entry is filled with the necessary virtual address bits as well as other information, and is inserted in the page table. This action allocates a *directory page* to the new page.

Page access information such as the reference and modify bits must be maintained. Since the DLB sees the virtual address stream after the attraction memory the reference bit in the DLB of a page with many coherence misses is updated much more often than the reference bit of shared read-only or non-shared pages. The impact on the accuracy of the replacement algorithm is hard to predict. The Modify bit in the DLB is implemented as follows. When a writable page is first created or loaded from disk, the Modify bit is set immediately if the page fault is caused by a write access. Otherwise, the state of the page's memory blocks is set to Master-shared. Then, if any node tries to get exclusive ownership for any attraction memory block of the page, the request is sent to the home node where the modification bits are set in both the DLB and the page table.

Besides segment-level protection, page-level protection can also be implemented in V-COMA. The protection bits must be copied in every level of the memory hierarchy. If a processor wants to change the protection bits of a page, it sends a message to the home node which hosts the page. The PE at the home node changes the bits in the page table and in the DLB. Then, according to the directory entries, it sends update messages to the nodes holding the blocks of that page. Another solution is to dedicate access right checking hardware within the processor, such as the PLB (Protection Lookaside Buffer) proposed for Single Address Space Operating Systems (SASOS)[18][46].

3.4 Experimental Evaluation

3.4.1 Methodology

We have run execution-driven simulations to compare the five options for dynamic address translation in COMAs. Our simulated baseline architecture has 32 nodes, each of which has a 200 Mhz Sparc processor. Because we can only simulate applications in which the data set sizes are much smaller than those expected for the target machine, we have to scale down the sizes of attraction memories, caches, and TLBs.

Each node contains 4 MB of attraction memory, a 16 KB first-level cache (FLC), and a 64 KB second-level cache (SLC). The FLC is direct-mapped and write through with a block size of 32 bytes. The SLC is 4-way set associative and write-back with a block size of 64 bytes. The attraction memory is also 4-way set associative, and its block size is 128 bytes. The page size is 4 KB for all simulations. The timing and network model are the same as in [53]. An FLC hit has no latency charge and an SLC hit takes 6 cycles. A hit in attraction memory takes 74 cycles. The network is an 8-bit wide crossbar clocked at 100Mhz. An 8-byte request takes 16 cycles and a message containing a data block takes 272 cycles.

We only simulate shared data accesses. Random replacement is used for the fully associative TLB/DLB. The parameters of the six Splash2 benchmarks[88] are shown in Table 3.1. The important working sets always fit in our simulated attraction memory, but sometimes do not fit in caches. The data sets fit in main memory and are preloaded so that no paging activity is simulated.

Benchmark	Parameters	Shared Memory (MB)
RADIX	-n524288 -r2048 -m1048576	6.12
FFT	-m20 -t	51.29
FMM	16384 particles	29.23
OCEAN	258*258	15.52
RAY-TRACE	car	34.86
BARNES	16384 particles	3.94

Table 3.1. Benchmarks

3.4.2 Address Translation Misses

Figure 3.9 shows the number of address translation misses per node as a function of the TLB/DLB size. The solid line for L2-TLB shows the number of misses in the case where SLC writebacks access the TLB. The L2-TLB/no_wback curve in dash line shows the number of L2-TLB misses without the writeback impact. Recall that we do not enforce inclusion between a TLB and the caches above it.

One obvious observation is that the number of address translation misses consistently decreases with the level of the TLB provided SLC writebacks do not access the TLB in L2-TLB. This is due to a *filtering effect* by the caches. Namely, the number of misses in a TLB cannot be larger than the number of misses in the cache above it. This effect is especially large when the TLB reach is less than the working set size. SLC writebacks affect the number of L2-TLB misses significantly, especially for FFT and OCEAN. With the writebacks, L2-TLB is sometimes even worse than L0-TLB. As explained before this is due to the poor locality of writebacks. Thus, it might be preferable to keep physical pointers in a virtual SLC so that writebacks can bypass the TLB.

The case of RADIX stands out (note the different scale in the graph). The curves show no clear significant working set for any TLB organization or size, until the size reaches 512 entries. RADIX has a disproportionately large number of writes and these write accesses cause coherence transactions and are not filtered by the caches or the attraction memory. Except for RADIX, the TLB-miss curve for L3-TLB is much flatter than for L2-TLB.

As we expected, the number of DLB misses is negligible for all benchmarks, even for very small DLB sizes. This is due to a *sharing effect.* DLB entries are shared and are not replicated. Thus the effective amount of DLB entries increases proportionally with the number of processors. This is not true for systems with TLBs. This effect can be huge, as in RADIX. In each pass of RADIX, a key is written into a large output array shared and distributed among all nodes. The number of DLB misses in RADIX is consistently less than the number of TLB misses in an L3-TLB system with 32 times more TLB (recall that we simulate 32 processors). All other bench-

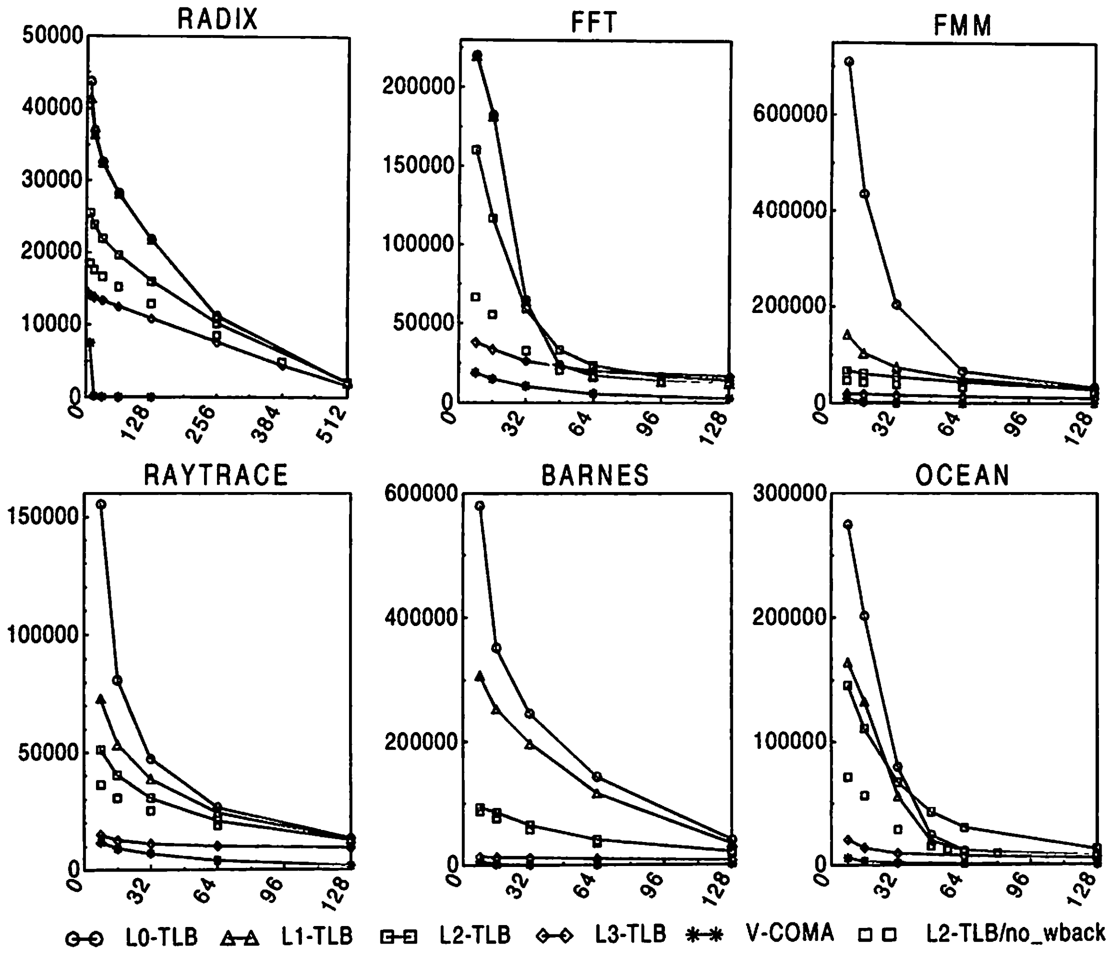


Figure 3.9. Number of Address Translation Misses vs. TLB/DLB size

marks show similar trends, albeit not as pronounced because their access patterns are more complex.

Since, in the case of RADIX, the DLB miss rate is lower than the miss rate of a TLB whose size is 32 times larger, another effect is at play besides the sharing effect. For example, a 16-entry DLB has even less misses than a 512-entry TLB in L3-TLB. This comes from a *prefetching effect*, a consequence of the sharing of DLB entries. Although the nodes do not interfere with the TLBs of other nodes in L3-TLB, they can not benefit from each other either. For example, if processor 1 writes a shared data which is then read by processor 2, a TLB miss at processor 1 does not help prevent a TLB miss at processor 2. The impact of this prefetching effect is significant for cold misses when the whole working set fits in TLB or DLB. In this case, every page table entry is loaded only once in the whole system in V-COMA instead of once per node in L3-TLB.

Table 3.2 shows the miss rates (misses/processor reference) for the DLB and TLBs. In L0-TLB the TLB miss rates are comparable to SLC miss rates when the TLB has 8 or 32 entries. Thus the TLB effects cannot be ignored. The situation improves somewhat in L2- and L3-TLB. V-

COMA is the only case where we could neglect address translation misses as compared to cache misses.

TLB/ DLB size	8					32					128				
SYS-TEM	L0	L1	L2	L3	V-COMA	L0	L1	L2	L3	V-COMA	L0	L1	L2	L3	V-COMA
RADIX	10.8	10.2	6.31	3.48	1.84	8.06	8.03	5.43	3.30	0.02	5.39	5.39	3.96	2.67	0.01
FFT	2.02	2.01	1.47	0.35	0.17	0.59	0.59	0.54	0.24	0.10	0.11	0.11	0.13	0.15	0.03
FMM	8.44	1.68	0.80	0.24	0.11	2.43	0.89	0.65	0.21	0.01	0.40	0.36	0.35	0.13	.004
RAY-TRACE	2.23	1.05	0.74	0.22	0.17	0.68	0.55	0.44	0.16	0.10	0.19	0.19	0.18	0.13	0.02
BAR-NES	2.68	1.42	0.43	0.06	0.03	1.13	0.91	0.30	0.05	0.0001	0.18	0.16	0.10	0.03	0.0001
OCEAN	6.45	3.86	3.42	0.48	0.14	1.87	1.32	1.58	0.23	0.04	0.16	0.16	0.30	0.12	0.003

Table 3.2. TLB/DLB Miss rates Per Processor Reference (%)

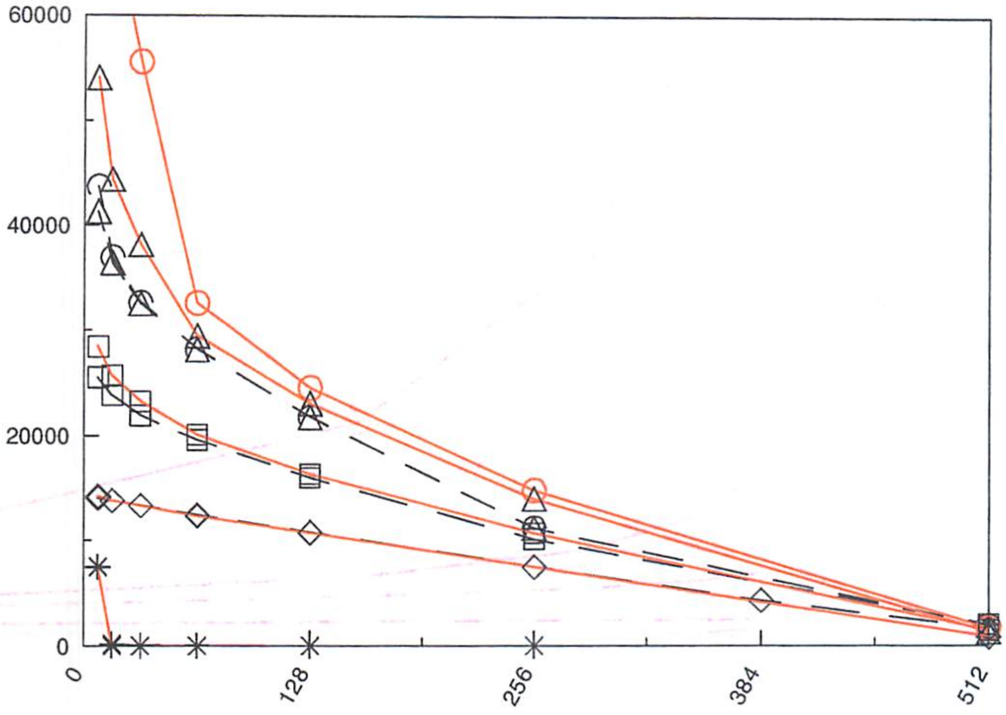
Another way to compare the five options is to ask which TLB size it would take to have the same performance as the DLB in V-COMA. The answer to this question is shown inTabl e3.3. We see that in many cases it would take TLBs with several hundred entries to have the same miss rate as an 8-entry DLB in V-COMA. Because of the writebacks, some equivalent L2-TLBs are larger than L1-TLBs, or even L0-TLBs.

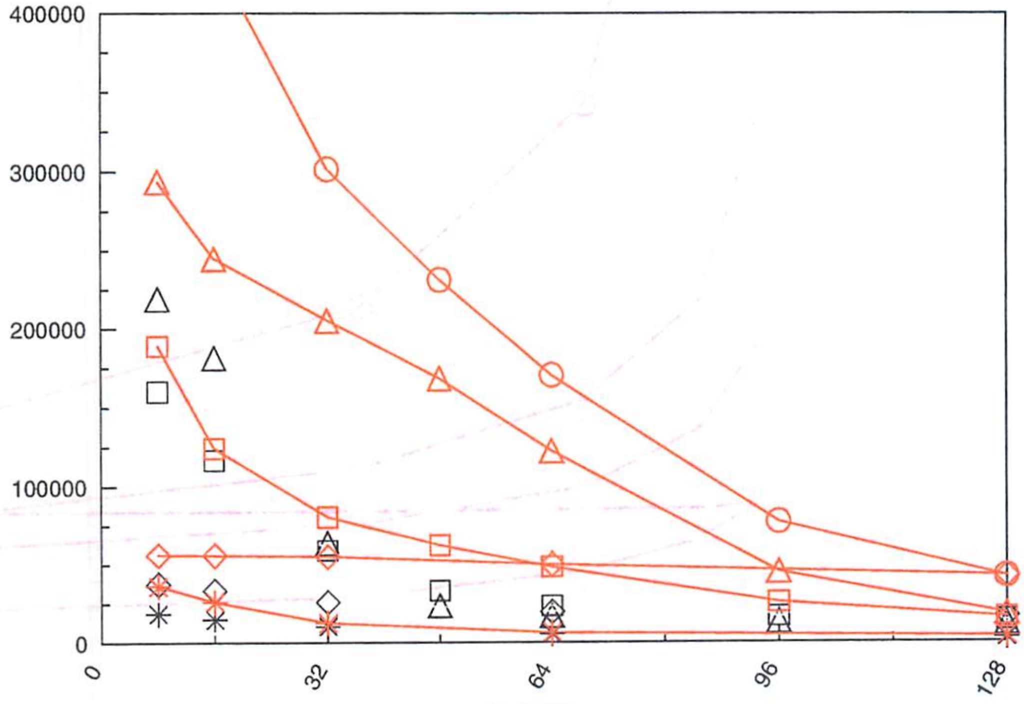
	L0-TLB	L1-TLB	L2-TLB	L3-TLB
RADIX	360	360	344	256
FFT	60	60	86	86
FMM	335	321	347	187
RAYTRACE	157	152	144	27
BARNES	327	318	298	160
OCEAN	175	174	251	113

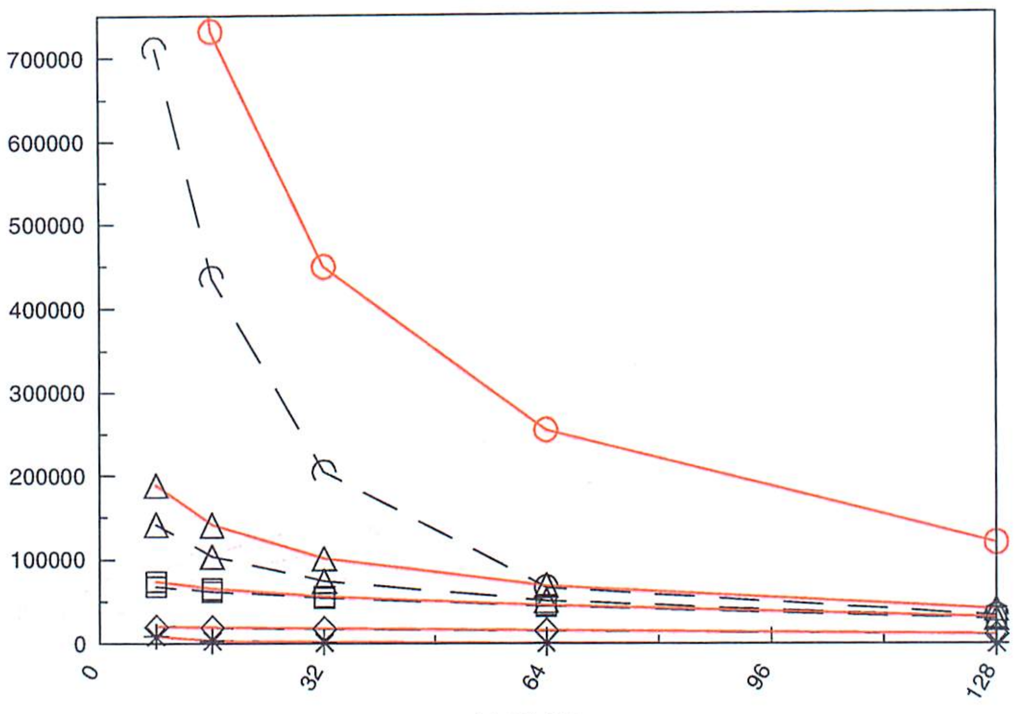
Table 3.3. TLB Size Equivalent to an 8-entry DLB

3.4.3 Sensitivity

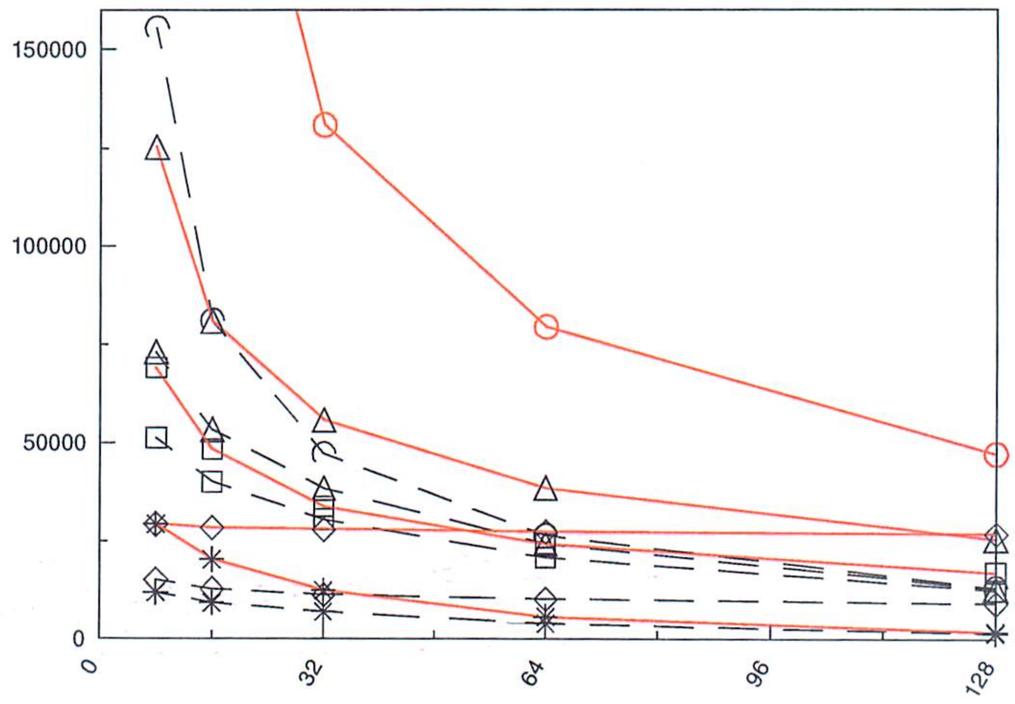
So far we have considered fully associative TLB/DLBs. However large, fully associative memories are slow and expensive. We also have simulated direct-mapped TLB/DLBs. Figure 3.10 shows the number of misses for the direct mapped vs. fully associative address translation mechanisms. The dash lines in the figure are for the fully associative cases which are the same as in Figure 3.9. The size and the set associativity of the virtual address caches can absorb potential


60000
40000
20000
0
0
128
256
384
512
(a) RADIX


400000
300000
200000
100000
0
0
32
64
96
128
L0-TLB
L1-TLB
L2-TLB
L3-TLB
V-COMA
L0-TLB/DM
L1-TLB/DM
L2-TLB/DM
L3-TLB/DM
V-COMA/DM
(b) FFT

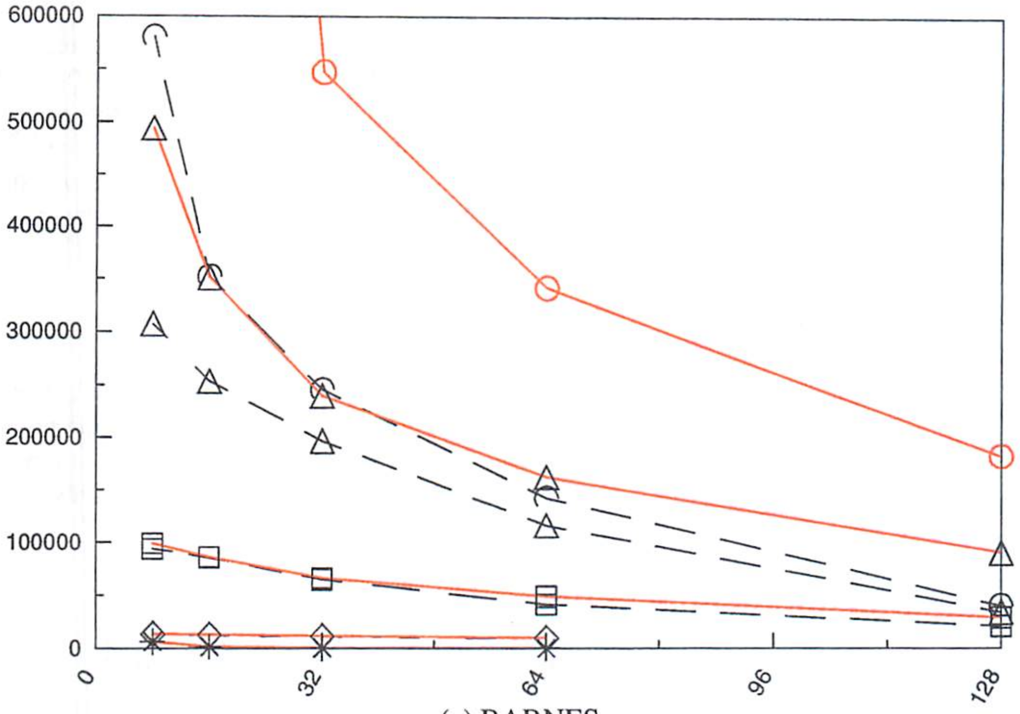


(c) FMM

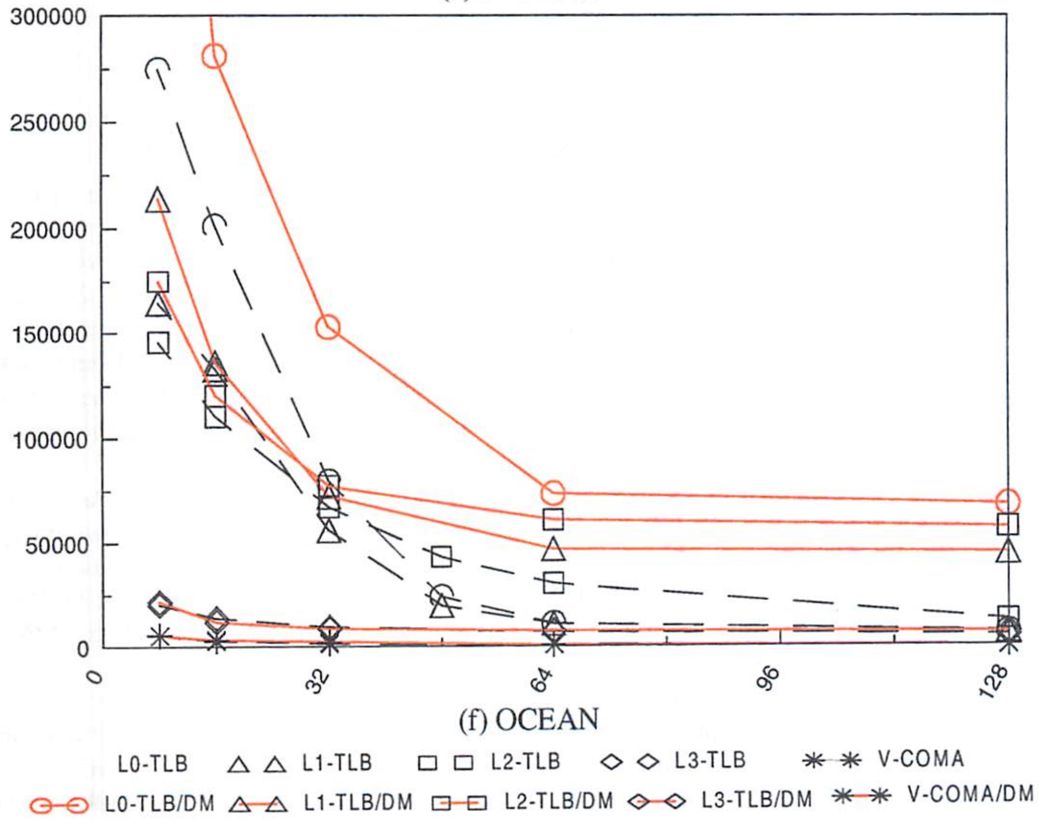


(d) RAYTRACE

L0-TLB △ △ L1-TLB □ □ L2-TLB ◇ ◇ L3-TLB ✳ ✳ V-COMA
○—○ L0-TLB/DM △—△ L1-TLB/DM □—□ L2-TLB/DM ◇—◇ L3-TLB/DM ✳—✳ V-COMA/DM



(e) BARNES



(f) OCEAN

Figure 3.10. Address Translation Misses for Direct Mapped TLB/DLB

conflicts which would happen in the TLB below it. As observed, whereas the gap between direct-mapped and fully associative TLBs is large for L0-TLBs and L1-TLBs, it becomes quite small in L2-TLB, L3-TLB and even more so in V-COMA. The big gap between the L0-TLB/DM and L0-TLB makes direct mapped TLB an unpractical design, and actually no existing processor adopts it. The *filtering effect* of the direct mapped write-through FLC is not as good as that of the larger set associative write-back SLC, while the *sharing effect* for the DLB helps reduce the gap further because the DLB coverage grows very fast with the DLB size per node. The large coverage of the DLB makes the organization less important.

3.4.4 Execution Time

We compare the execution times of V-COMA with those of the L0-TLB scheme, a traditional physical COMA. Physical addresses are distributed round robin among processors. Based on the timing of our simulation model and the average TLB service times in current computer systems[54][79], 40 cycles are charged for each L0-TLB miss. Each DLB miss also takes 40 cycles. The memory consistency model is sequential consistency.

	L0-TLB (8)	DLB (8)	L0-TLB (16)	DLB (16)
RADIX	10.61	1.25	8.93	0.04
FFT	15.24	0.88	12.56	0.76
FMM	96.54	1.15	59.54	0.38
RAY-TRACE	30.95	1.04	17.46	0.82
BARNES	38.14	0.45	22.12	0.01
OCEAN	21.53	0.45	15.95	0.23

Table 3.4. Address Translation Time / Total stall time (%)

Table 3.4 shows the average TLB/DLB overhead divided by the average processor stall time on local and remote memory accesses for DLB/TLBs of sizes 8 and 16. Note that these small TLB sizes have been selected to offset the effects of the small data set sizes in our benchmarks. The data show that address translation is a significant part of the memory latency in the traditional L0-TLB system and that its effect is at least comparable to the effect of memory consistency models [28]. Fortunately, the memory access penalties due to translation can be drastically cut to a point where they are negligible by translating addresses at the home node.

The effects on the execution time are shown in Figure 3.11, where *Busy* indicates the time spent executing the instructions in each processor, *Sync* is the synchronization time, *Loc-stall* counts time spent on local cache misses, and *Rem-stall* refers to the service time for attraction memory misses. TLB/8 is L0-TLB with an 8-entry fully associative TLB, DLB/8 is V-COMA with an 8-entry fully associative DLB, TLB/8/DM and DLB/8/DM correspond to the same systems but with direct mapped TLB/DLBs.

Address translation overhead is negligible in V-COMA, as we expected. The total execution time improvement over the physical COMA depends on the original address translation overhead. Virtual address indexing of attraction memory does not have a significant impact on the overall execution times (excluding the TLB overhead), although in general it is no better than for the physically indexed attraction memory except for BARNES. In RAYTRACE however we

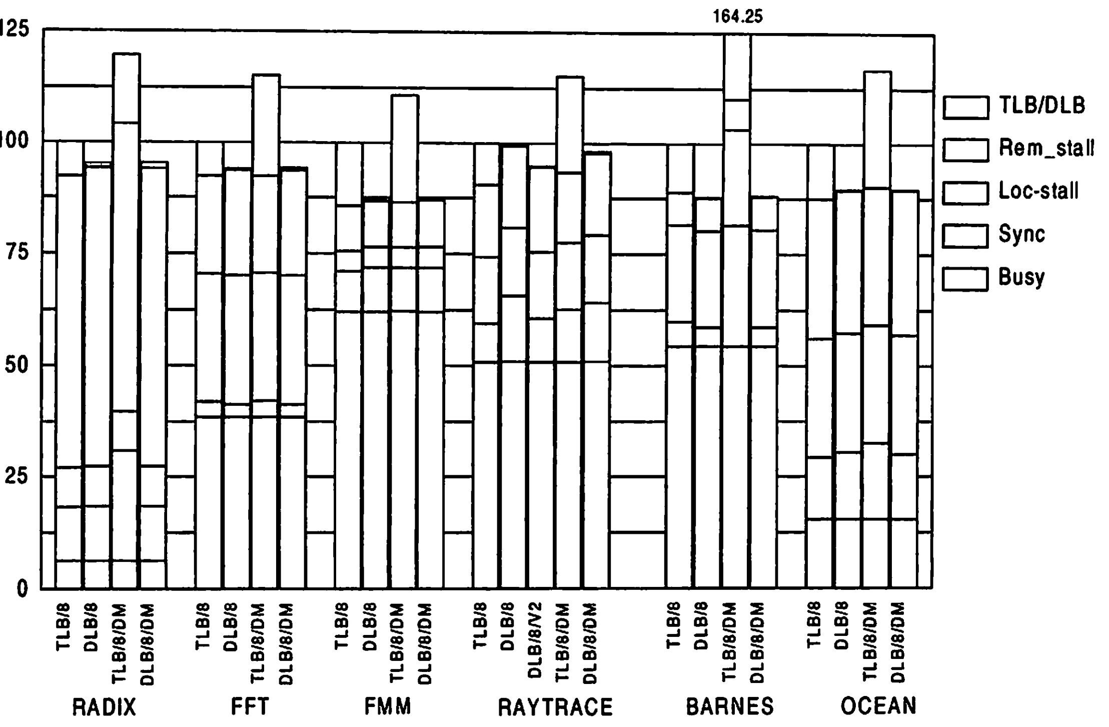


Figure 3.11. Execution Time of V-COMA and L0-TLB

observe that the execution time of V-COMA (excluding the TLB overhead) is much larger than in physical COMA due to the increased synchronization time. This effect is due to the fact that the virtual address layout is not optimized, whereas the round robin pageframe assignment turns out to be a good strategy for the physical COMA. RAYTRACE shows an extreme case of this effect because in the definition of *raystruct*, which is the private stack for the ray tree of each node, padding is used to avoid false sharing. The padding is aligned on multiple of 32KB in the virtual address space, which causes uneven conflicts in V-COMA leading to the increase of the synchronization time. By simply aligning the padding to one page size (4 KB), the synchronization time is reduced significantly, as shown as DLB/8/V2 in Figure 3.11. This simple example indicates that there is plenty of room for virtual address layout optimization for the V-COMA architecture.

3.4.5 Relative Miss Rate and TLB-less Systems

The TLB/DLB miss rate is decreased when it is moved further down the memory hierarchy as shown inTable 3.2, due to the filtering effect of the caches. The number of accesses to the TLB is much less when it is located after a cache than if it is located before the cache. Therefore, as the total number of TLB/DLB misses is decreased, so is the TLB/DLB miss ratio per processor reference. However as we have seen, this is mostly due to the filtering effect of the cache. The effectiveness of the TLB is decreased in general when it is moved further away from the processor because of the poorer locality of the memory reference stream after each cache. Table 3.5 and Figure 3.12 show the relative TLB/DLB miss rate, which is the percentage of TLB/DLB misses relative to the number of TLB/DLB accesses. Due to the sharing and prefetching effect, V-COMA exhibits much better relative miss rate than L3-TLB, which is also accessed after the attraction memory, and its relative miss rate improves much faster with the DLB size.

TLB/ DLB size	8					32					128				
SYS-TEM	L0	L1	L2	L3	V-COMA	L0	L1	L2	L3	V-COMA	L0	L1	L2	L3	V-COMA
RADIX	10.8	21.3	71.9	85.4	44.5	8.06	16.8	61.8	81.0	0.48	5.39	11.3	45.1	65.5	0.32
FFT	2.02	4.97	39.1	40.1	26.0	0.59	1.45	14.5	27.7	14.5	0.11	0.27	3.48	17.6	4.21
FMM	8.44	17.8	50.7	54.9	27.6	2.43	9.34	41.0	48.2	3.87	0.40	3.76	22.2	30.5	0.91
RAY-TRACE	2.23	5.02	23.8	28.7	20.0	0.68	2.65	14.2	21.6	11.8	0.19	0.89	5.79	17.6	2.50
BAR-NES	2.68	3.85	54.7	52.8	25.7	1.13	2.46	38.1	44.5	0.13	0.18	0.43	12.6	27.7	0.13
OCEAN	6.45	12.5	40.5	31.4	10.1	1.87	4.25	18.6	14.8	2.96	0.16	0.51	3.52	8.13	0.21

Table 3.5. TLB/DLB Relative Miss Rates (%)

Except for V-COMA, fast but small TLB in memory hierarchy is not as efficient as the traditional L0-TLB, with respect to the relative miss rates. Although the TLB performance is less sensitive to its organization and can be built much larger, another design choice is to do without the TLB. Due to the cache filtering effect and the poor efficiency, the TLB after a large virtual address cache can be eliminated without significant impact on system performance. Indeed, removing the TLB could save some hardware cost, and more importantly get rid of the overhead of maintaining TLB consistency, which does not scale well in multiprocessors. Actually, the in-cache translation[89] and software-managed translation[42] can be considered as L2-TLB schemes with 0 TLB entry.

The mechanism to search page table for address translation in case of virtual address cache misses is very important to maintain robust system performance when the TLB is eliminated. The virtual address translation can be served by trapping the main processor as in [42], by the protocol engine, or by a hardware controller through some programmable logic. Despite the possible caching for page table entries in all schemes, schemes other than V-COMA could have remote page table accesses, while in V-COMA, a page table is always accessed locally, which means that the time to find a translation is less than in other architectures.

For L2-TLB, L3-TLB and V-COMA, the performance impact of 0 TLB/DLB will not be significant. even though every coherence miss triggers address translations. Compared to the long latency of coherence activities, this address translation overhead could be very small. Removing L1-TLB will cause significant performance degradation due to the limited size and write through behavior of L1 cache. Although V-COMA has much smaller DLB miss rate and much higher effectiveness than L2-TLB and L3-TLB, removing the DLB should not hurt V-COMA badly because of the very low access frequency to the DLB.

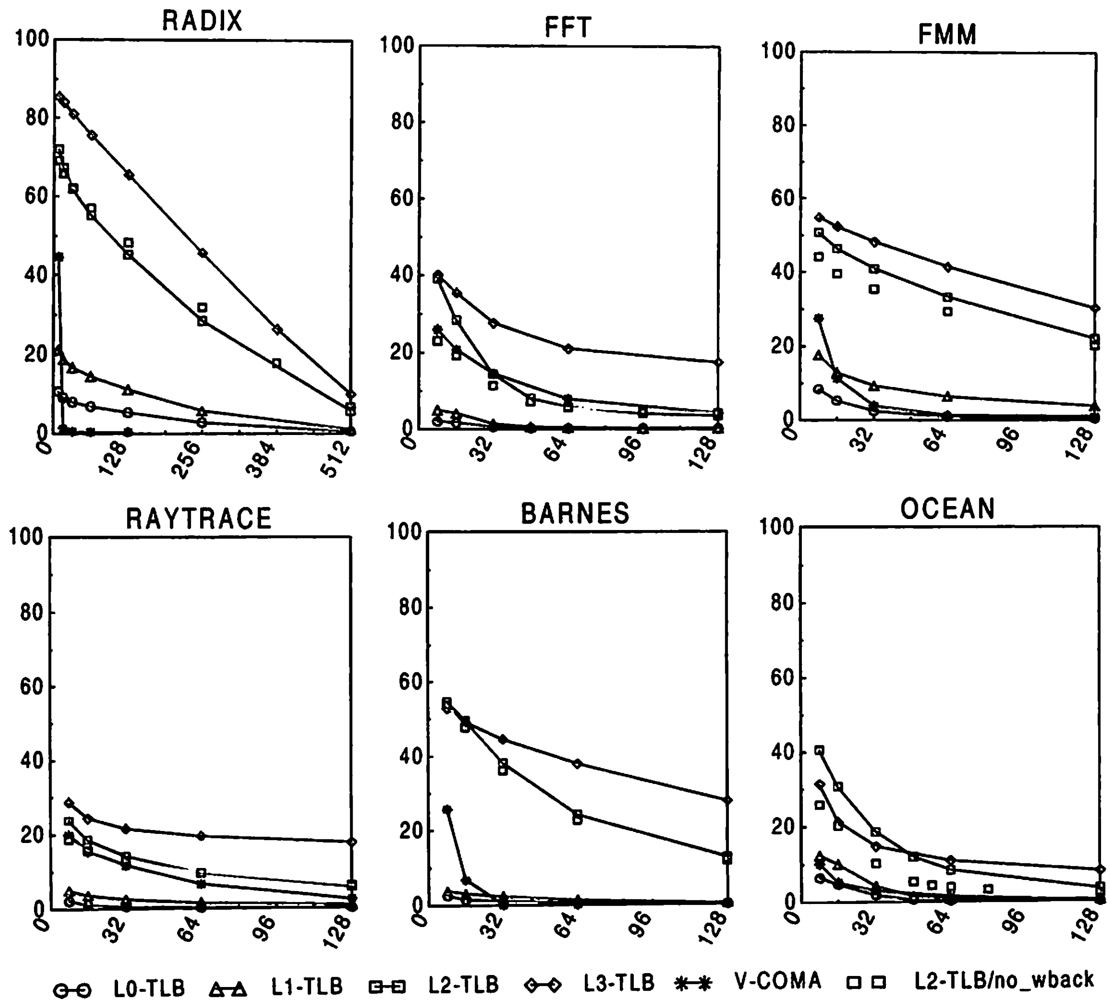


Figure 3.12. Relative TLB/DLB Miss Rate (%)

3.5 Further Discussion on Virtual COMA

Set Associative Memory. Although the V-COMA architecture eliminates address translation overhead, the restriction on memory mappings may be of concern. If the attraction memory at each node can contain N pages and is K-way set associative, the number of global page sets is N/K. Each global page set contains up to P*K page slots, where P is the number of processing nodes. The memory management unit in a traditional physical COMA can control the *global page set* index of a new virtual page through the virtual-to-physical page mapping. On the other hand, V-COMA has no control over the *global page set* index for the virtual pages. The danger is that the conflicts in the virtual address space will affect the page fault rate and thus overall system performance.

Although this danger is real, it can be overcome. First, the virtual address has good locality within each process[51]. Although virtual address layout tends to be uniform across different

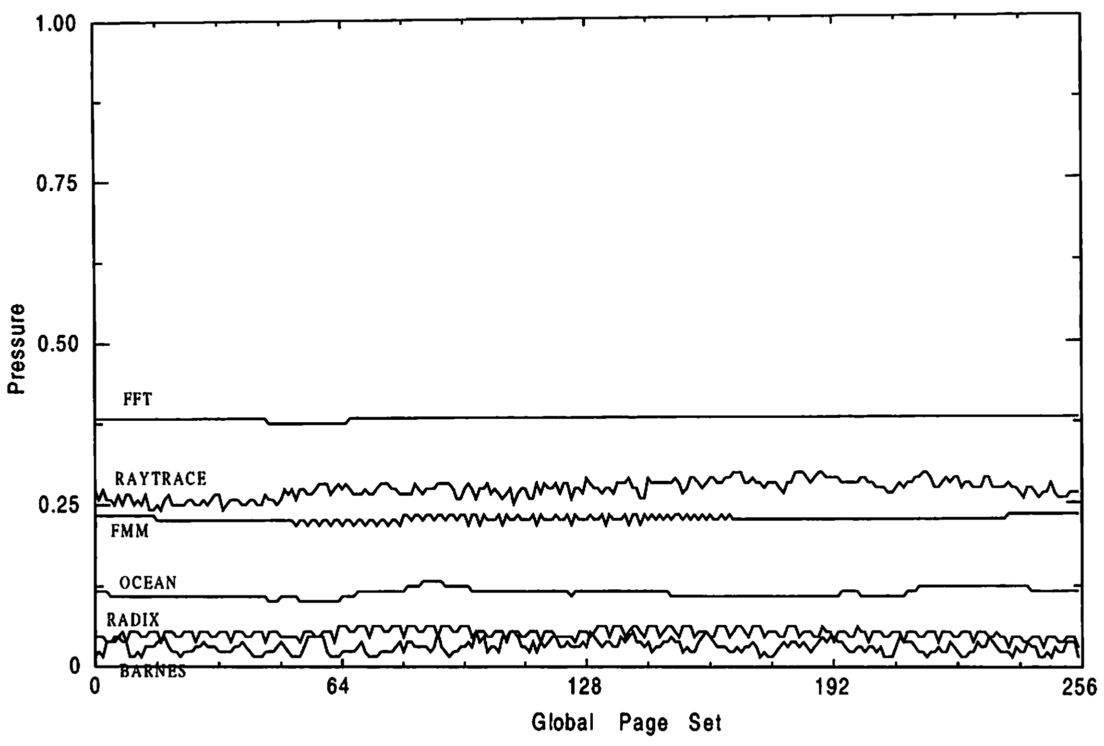


Figure 3.13. Pressure Profile for the Benchmarks

processes and may cause high conflicts in some *global page sets* and under-utilization of others, this situation can be improved by several methods. First a base address could be added to each virtual address; this base address can be different for every context. Second the code and data segments of different contexts could be located in different areas of their virtual space by the compiler[1]. Third, the operating system controls dynamic memory allocation, and can optimize the scheduling strategy to spread paging activity evenly across the *global page sets*. Fourth, the whole problem is a non-issue in large scale systems, considering that the capacity of each *global page set* scales up linearly with the number of processing nodes. For example, if each node has a 32MB 8-way set associative attraction memory, a 1024 processor system has up to 1024*8 = 8192 page slots per global page set, which is equal to the total number of physical address pageframes on each node!

Figure 3.13 shows the pressure profile on each global page set in our evaluation. It shows, that even without trying we observe a very uniform pressure on every global page set. This is because program locality is maximized in the virtual space. In the physical space, locality only exists within a page. Our experience with allocating memory in physical COMA [53] has taught us that the intervention of the memory management unit can result in disastrous --albeit well-intended-- physical page allocations for COMAs.

In fact, the advantages of this memory structure are very attractive. As a machine running on virtual addresses, V-COMA provides a simple and consistent hardware model to the operating system and the compiler where further optimization opportunities are possible. Virtual addresses are divided into S sets each of which has P*K page slots. The linear contiguous physical memory space no longer exists, and the management of the pages in the main memory is much simplified.

While the operating system guarantees correct execution, the performance of V-COMA is enhanced by the compiler and the programmer. Unlike in the traditional flat COMA where the programmer and compiler have no chance of affecting memory pressure and conflicts, the virtual address layout directly controls the occupation of global page sets in the attraction memory. Static virtual address layout optimization can generate good programs which get more performance benefit from the V-COMA architecture.

Virtual Address Tag. Because the virtual address is usually longer than the physical address, the tag memory is larger in V-COMA. For example, the 32-bit PowerPC architecture has 52-bit virtual addresses and 32-bit physical addresses; the 64-bit PowerPC architecture has 80-bit virtual addresses and 64-bit physical addresses. With the access right bits, the virtual tag may be 2 to 3 bytes longer than the physical tag. This will increase the tag memory by a small fraction of the attraction memory. If this is a concern in a particular design, techniques to reduce tag sizes can be considered[73][86].

Extensions and Optimizations. Due to its special architecture, V-COMA offers the opportunity to build a more efficient and smarter memory system. The following are some examples of extensions and optimizations of the V-COMA architecture.

- **Smart Replacement.** V-COMA offers an opportunity for a smart replacement policy compared to other flat COMA protocols. Because the home node assignment bits are part of the bits of the global page set index, all the attraction memory blocks within each global set will be assigned to the same home node. The disadvantage of this is that it will reduce the probability of a replacement block to be accepted by the home node because all the replacement messages within each global set are sent to the same home node. However, because the home node contains the directory information for all the blocks within the global set, it can find out which nodes have vacancy to accept the replaced block, and even predict among them which one will most probably access the data in the future based on access history. These optimizations can reduce both the replacement traffic and the node miss rate.

- **Moving Functions to Memory.** The page tables are distributed among the home nodes and under the control of the protocol processor. Therefore, it is possible that some operating system functions, especially virtual memory related functions, can be migrated to the protocol processor. For example, periodical updates of the reference bit for the page table entries can be done easily by the protocol processor. Some system processes such as *fsflush* which periodically synchronizes the file system between memory and secondary storage can also be moved to the memory side instead of executing in the main processors. If the protocol processor is powerful enough, we can further imagine that virtual memory system or even the whole operating system may move to memory side because the protocol processor can understand virtual addresses. Besides the operating system functions, it is also possible that some user level functions can be moved to the memory side given that the processing bandwidth is enough. A good example is database server daemons.

- **Fast Page Fault and Victim Cache at Home Node.** One of the drawbacks of V-COMA is the set associative main memory. Unlike traditional systems in which the page table implements a fully associative mapping for main memory, V-COMA depends on the hardware caching mechanism to locate the data and therefore inherits the set associativity of the attraction memory. We have discussed that it does not have significant impact on the overall system performance. Moreover, the impact can be further reduced by reserving some main memory at the home node to be shared by all the global sets on the node as a victim cache. This reserved main memory can be managed by the operating system as page caches. Access to these pages will generate fast page fault which is handled by the protocol processor at the home node. Alternatively, the reserved memory can also be managed without OS involvement as a victim cache for the global sets of attraction memory. When receiving a replacement message, the home node can make the decision whether to keep the block in the attraction memory and inject it into some node or kick it out of the attraction memory and reserve the block in the local victim cache.

- **Easy System Extension for V-COMA.** Unlike other software DSM systems where the operating system has to allocate physical addresses for the shared memory of remote node, V-COMA can be much easier to expand beyond the pure hardware limit. Physical addresses do not exist in V-COMA. Data blocks identified by virtual addresses are injected into the attraction memory without the need to allocate a physical address. Since virtual memory implementation is integrated with the cache coherence protocol, caching for outside data is much more flexible and does not need to be backed up by physical memory. The protocol processors can make local decisions for caching and swapping based on the global sets of the attraction memory.

3.6 Summary

The virtual memory implementation in current computer systems does not scale well for various reasons. The dynamic virtual address translation of the TLB is becoming a performance bottleneck with the current trends of the technology and the applications. Virtually-addressed memory hierarchies move the TLB out of the processor core and locate it in the memory hierarchy.

In this chapter, we have analyzed and evaluated five different designs for virtually-addressed memory hierarchies in the context of large scale distributed shared memory multiprocessors. The dynamic address translation mechanism is located after the caches and the caches are virtually indexed and virtually tagged. We have discussed various issues related to virtually-addressed memory hierarchies in multiprocessors. We have run detailed execution-driven simulations to evaluate the systems. Our simulation results have shown that the overhead of the dynamic address translation is dramatically reduced when it is moved down to the memory hierarchy due to the filtering effect of the virtual address caches.

We have proposed a novel multiprocessor architecture called V-COMA, which extends the concept of virtual address cache to its limit so that the entire memory hierarchy is virtually-addressed. While still supporting a conventional virtual memory system, V-COMA reduces the address translation overhead to a minimum. The traditional physical address concept is eliminated in V-COMA and the virtual memory support is integrated with the cache coherence protocol. The DLB in V-COMA, which translates virtual address into directory address at the home node, is shared among all processor and scales with the main memory. Therefore, in addition to the filtering effect, the DLB also benefits from a sharing effect and a prefetching effect so that the address translation overhead becomes negligible. Although the virtual memory has a set-associative organization, V-COMA scales well and works better in larger-scale systems. As a machine running on virtual address, V-COMA provides a simple and consistent hardware model to the operating system and the compiler in which further optimization opportunities are possible.

Chapter 4

VIRTUAL ADDRESS CACHE

In the past, the strong motivation to remove some of the TLB overhead drove several companies to adopt virtually-addressed caches (i.e. caches indexed and tagged with virtual addresses a.k.a V/V caches) and inspired some very ambitious academic research projects in both software and hardware such as the Opal operating system project [18] at the University of Washington and the Spur project [89] at Berkeley.

However, the well-known "curse" of virtually-addressed caches is the synonym problem: because of sharing and other factors, multiple virtual addresses may map to the same physical address. This complicates cache management because of the lack of a system-wide unique identifier for memory locations. Unique identifiers are also required to disambiguate memory addresses in the load/store unit of ILP processors.

One approach to create unique identifiers is segmentation[15][42][52][89]. Segment registers managed by software translate user virtual addresses into global virtual addresses before accessing the cache. An example of a segmented architecture is illustrated as Figure 4.1. Sharing is implemented at the segment level. Segments have fixed size and very coarse granularity. The operating system must support the global virtual addresses and the segment structures and the porting effort is large, requiring ad-hoc patches for various situations involving synonyms[15].

Another approach is the Single Address Space Operating Systems (SASOS)[18], where all processes share one single address space. Synonyms are resolved by using the same virtual address. However, the software community is clearly not ready for such a radical change.

This chapter introduces new ideas to enable the use of virtual addresses throughout the memory hierarchy, thus removing the TLB from the processor and associating it with memory where it scales much better. The major idea is the replacement of the TLB in the processor by a Synonym Lookaside Buffer (SLB), which can remain small because its size depends on the number of synonyms, not on the size of the application or of the physical memory. The synonym lookaside buffer(SLB) is a translation cache very much like a traditional TLB but it only translates synonyms. It creates unique identifiers as the segment registers in the PowerPC architecture[52], but it is much more flexible and is managed in hardware without affecting the software. Because the mapping is more flexible and is restricted to synonyms, it is very effective even if the number of SLB entries is small. The SLB can be accessed before or in parallel with the first level cache.

4.1 Synonyms

Multiple virtual addresses mapped to the same physical address are called *synonyms*. Synonyms are very convenient to kernel and user software in many situations. Synonyms should be aligned at least on page boundaries since virtual memory is managed in page granularity.

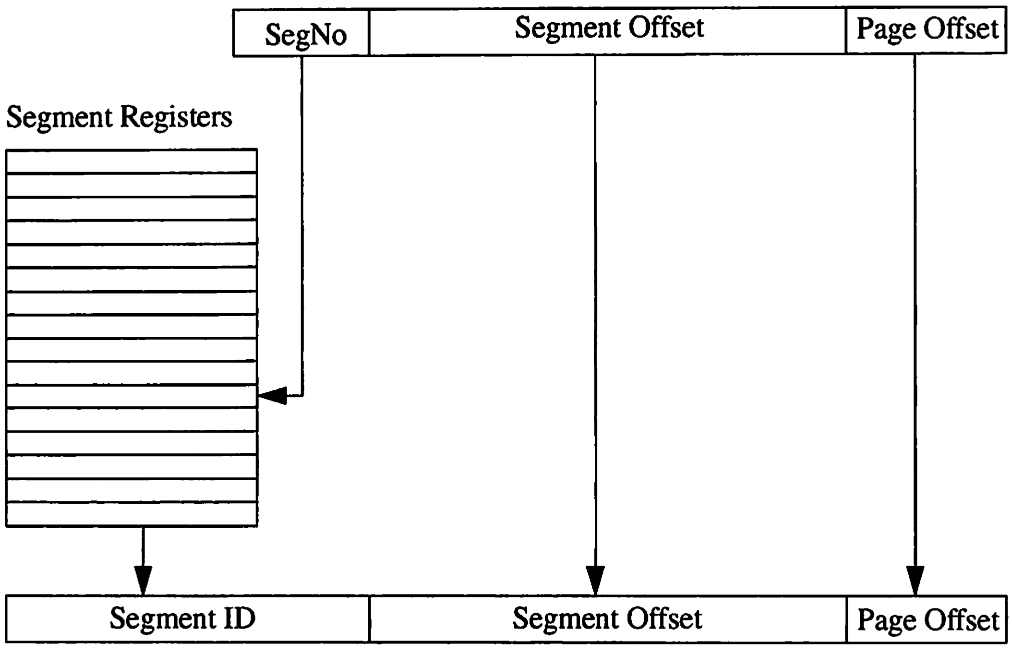


Figure 4.1.Segment Translation

Synonyms are often used to implement shared memory semantics across different user virtual address spaces. Read-only segments such as libraries and text segments are widely shared and the kernel is usually shared among all private virtual address spaces at a fixed location. User processes can also request shared memory segments to facilitate communication with other processes. Users may even define synonyms within the same process for convenience.

In addition to the true sharing semantics, synonyms are also critical for the efficiency of various memory operations. Copy-on-write avoids unnecessary memory copies and reduces the consumption of physical memory. In this case, different virtual addresses share the same physical memory location for as long as the content is not modified.

It is very frequent that a particular physical page P is remapped from one virtual page V1 to another virtual page V2. We consider that V1 and V2 are synonyms even though the V1-P and V2-P mappings may not coexist at the same time. Traditional message passing semantics can be optimized using the remapping operation to avoid real memory copies. As an example, process 1 wants to send a message page at virtual address V1 to process 2 at virtual address V2. Instead of copying V1 to V2, the system can simply remap the physical page P from V1 to V2. The old V1-P mapping can either remain valid as copy-on-write or destroyed depending on the message semantics. As a special case, the operating system kernel usually buffers the pages used in I/O in physical memory accessed with kernel address and remap them to user address space when needed.

In the message passing optimization, it is the content of the physical page P that is passed from V1 to V2. However, it is also very common that V1 and V2 share the resource of the page-frame rather than its content. For example in demand paging system, V1 may be swapped out and the physical page P can be freed and allocated to another virtual page V2. V1 and V2 do not have any logical connection. Physical page P should be overwritten with new content before it is allocated to V2.

4.2 Virtual Address Caches

4.2.1 Translation Lookaside Buffer (TLB) and V/P Cache

Figure 4.2 shows a typical architecture for a V/P cache and a TLB accessed in parallel. Cache blocks are named by their physical address, while their virtual address is used as a hint or hashing function to quickly find their location. After a cache set is indexed by the virtual address, all tags in that set are compared with the physical page number translated by the TLB. A miss in that set does not guarantee that the data block is not present in the cache. The block may reside in any set within the superset. The superset is made of all the cache sets whose set indexes share the same (p-b) least significant bits, where p is the number of page offset bits and b is the number of block offset bits.

To detect synonyms in the superset a backpointer[85] is stored in the L2 physical-address cache for every valid block in the L1 cache. Each pointer points to the frame in L1 where the block is valid. A synonym is detected whenever an L1 cache miss hits in the L2 cache and the backpointer is valid in L2. In this case, the block is moved to the new cache set in L1 and retagged, and the backpointer in L2 is updated. This is called a *short miss.*

When the TLB misses, usually the processor is trapped and the TLB is filled by the trap handler before resuming the access. The miss rate of the TLB is primarily determined by its reach or coverage, which is the aggregate virtual memory area mapped by all its entries[19]. Superpages can improve TLB coverage[79]. However, if one TLB entry maps a superpage, all the physical pages inside the superpage must be allocated in contiguous frames in physical memory. The size of a superpage is also hard to figure out at the time when the virtual address is first allocated. Page placement in multiprocessors and coloring of pages to optimize cache behavior[7] may conflict with the use of superpages.

4.2.2 Synonym Lookaside Buffer and V/V cache

To solve the synonym problem, we must associate a unique identifier to each page. To do this, we split the entire virtual address space in two sets: the main address set and the synonym set. The synonym set contains all the virtual addresses that are synonyms to some virtual addresses in the main address set. The main address set contains the rest of the virtual addresses which include virtual addresses with no synonyms and one virtual address selected in each group of synonyms. The main addresses act as unique identifiers for all pages.

An SLB (Synonym Lookaside Buffer) dynamically translates virtual addresses in the synonym set into their corresponding shared virtual addresses in the main address set. Just as a traditional TLB, the SLB can be accessed in parallel with the first level cache. Figure 4.3 shows the parallel access to cache and SLB. Backpointers in the second-level cache take care of short misses

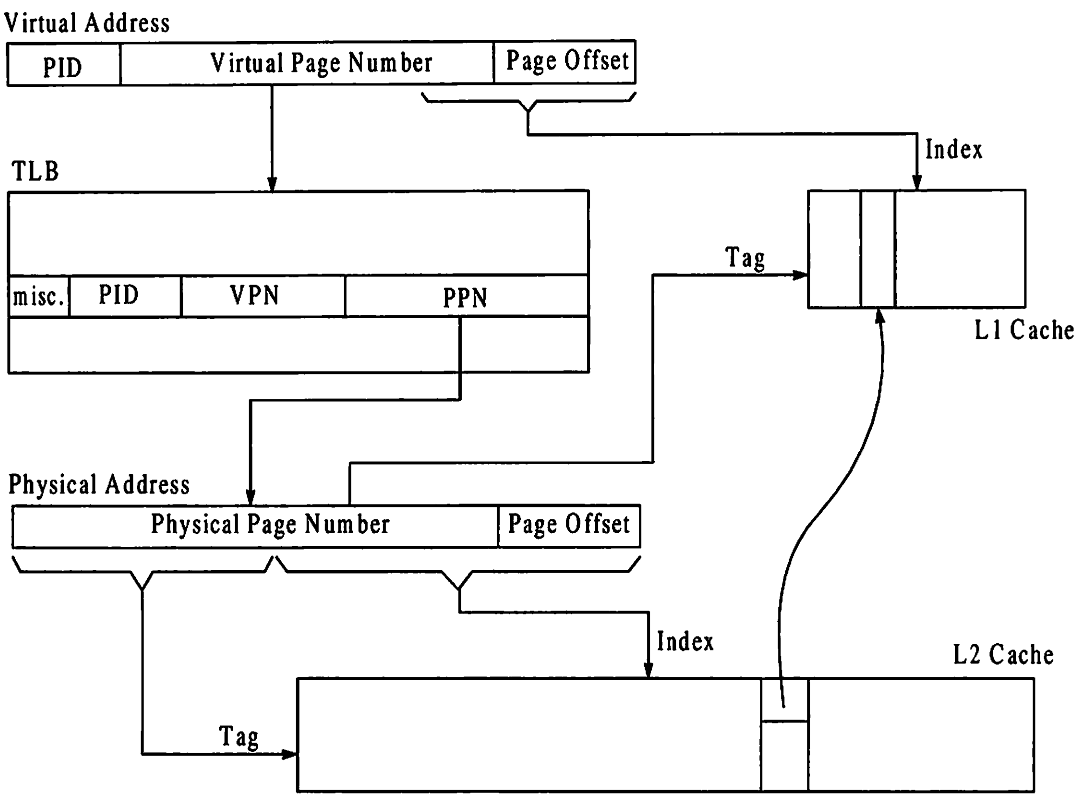


Figure 4.2. TLB and V/P Cache

as in the V/P cache. The main difference between Figure 4.2 and Figure 4.3 is that each SLB entry contains the main address instead of the physical address.

As we will see, a small SLB of 8 or 16 entries (similar to the number of segment registers in segmented architectures) is sufficient. It is therefore possible to access the SLB on the access path to the first-level cache. To be protected against programs with bad behavior, we can imagine adding a second level SLB accessed in parallel with the L1 cache. In case a synonym misses in the first-level SLB but hits in the second-level SLB, the cache access is aborted and reissued using the main address translated in the second-level SLB. When the SLB is accessed before the L1 cache, L2 cache backpointers and short misses in L1 cache are eliminated.

A virtual address missing in the SLB may be either a synonym or a main address. Whether or not it misses in the SLB the memory access proceeds. If it is a synonym, it will miss in the cache hierarchy and will eventually be detected at the point where the physical address is needed, trapping the processor which then fills the SLB. The access is then retried using the main address.

Just as in the traditional TLB, the SLB entry must include access right bits corresponding to the synonym address. The cache hierarchy also contains protection bits associated with the main address. A synonym access is checked for protection twice, once in the SLB and once in the cache hierarchy. The main address must have equal or less access restrictions than all of its synonyms.

Most current operating systems implement demand paging virtual memory. On top of the machine-dependent layer, there is usually a machine-independent layer organizing virtual memory in logical segments. Sharing of synonyms is decided in the logical layer independently of physical

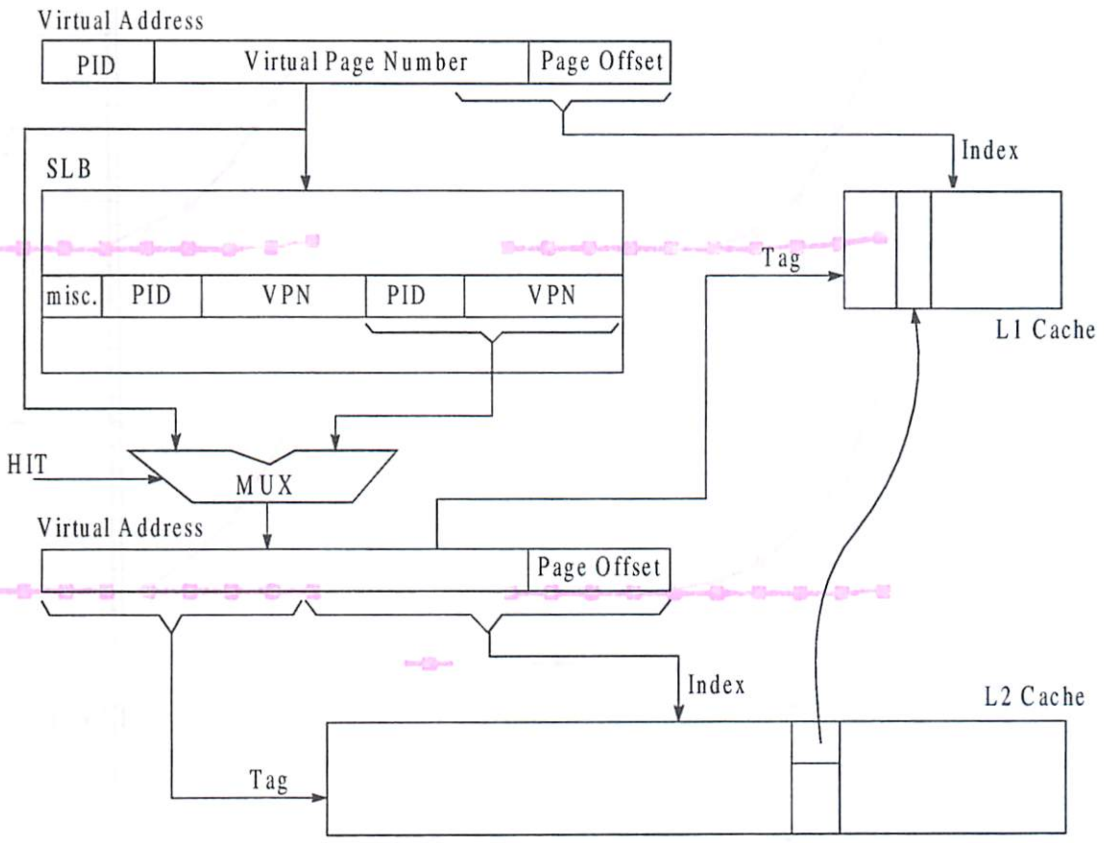


Figure 4.3. SLB and V/V Cache

paging. Since sharing is already managed in the operating system in the logical layer, only some minor changes are needed in current operating systems to use an SLB.

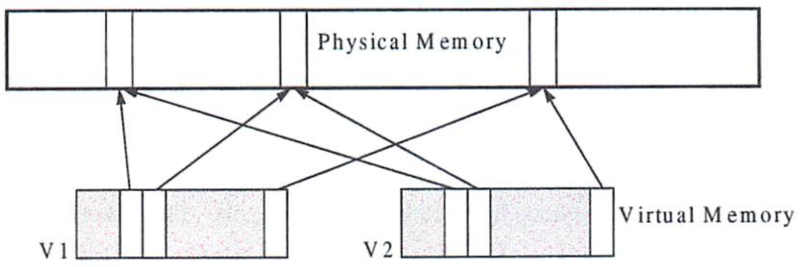


Figure 4.4. A Synonym Example

4.2.3 Comparing TLB and SLB

Although the SLB has practically the same hardware organization as a traditional TLB, its scalability is much better. Some issues specific to SLBs must also be solved.

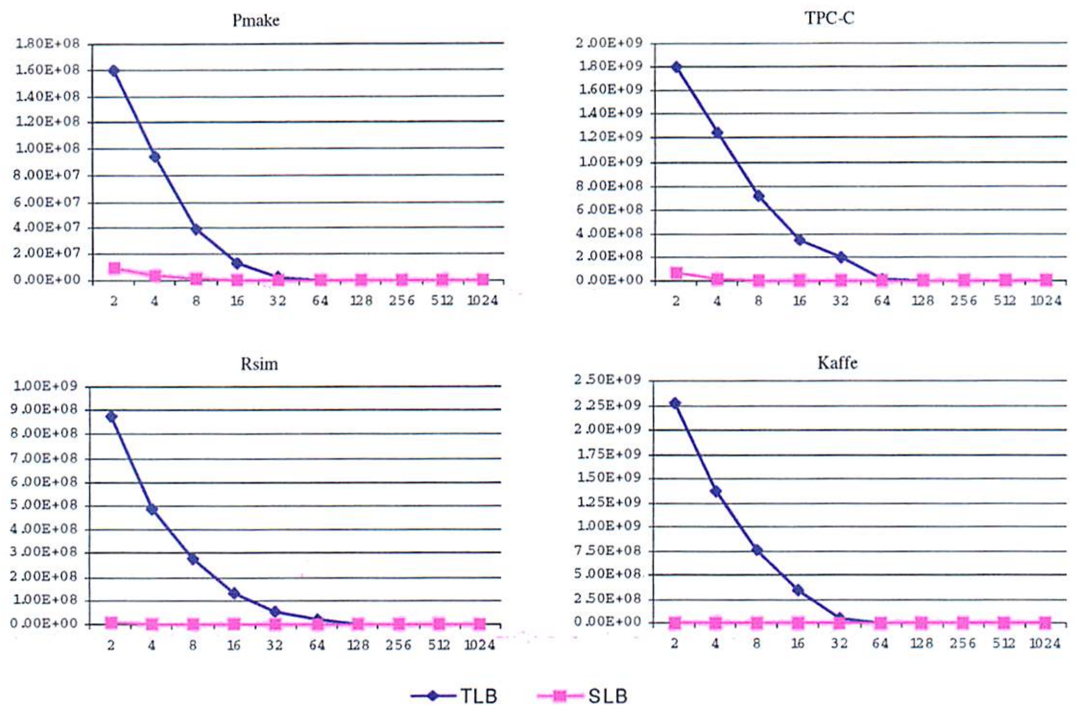


Figure 4.5. Total Number of Misses as a Function of the Number of Entries in a TLB and in an SLB for Four Benchmarks

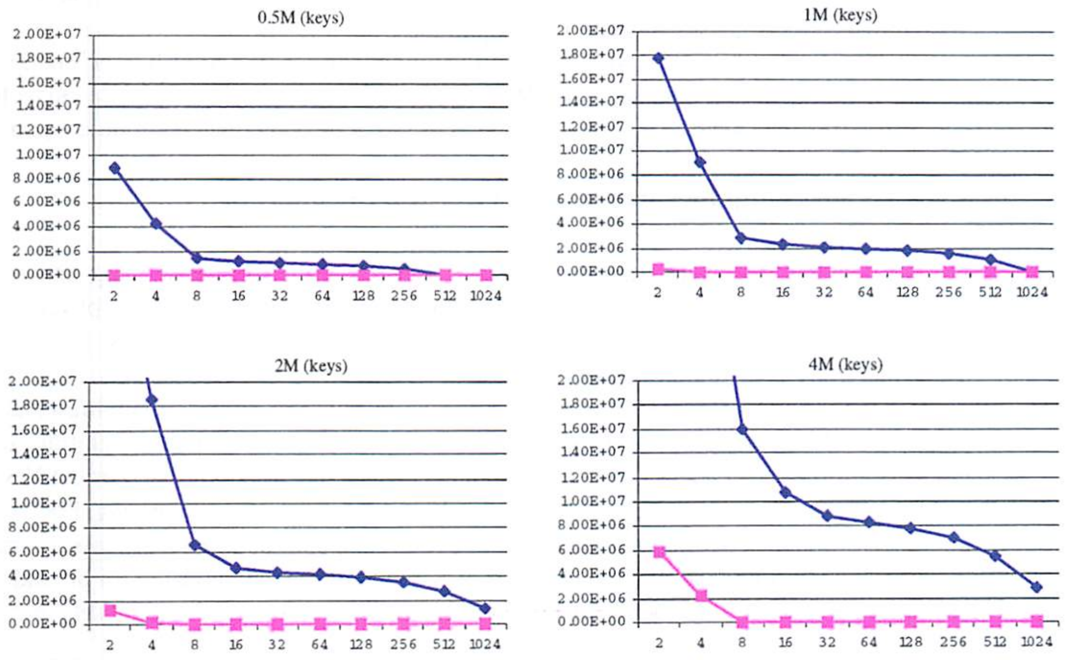
4.2.3.1 Coverage

The coverage of the SLB is much better than that of a TLB and scales with application and main memory sizes, for two major reasons.

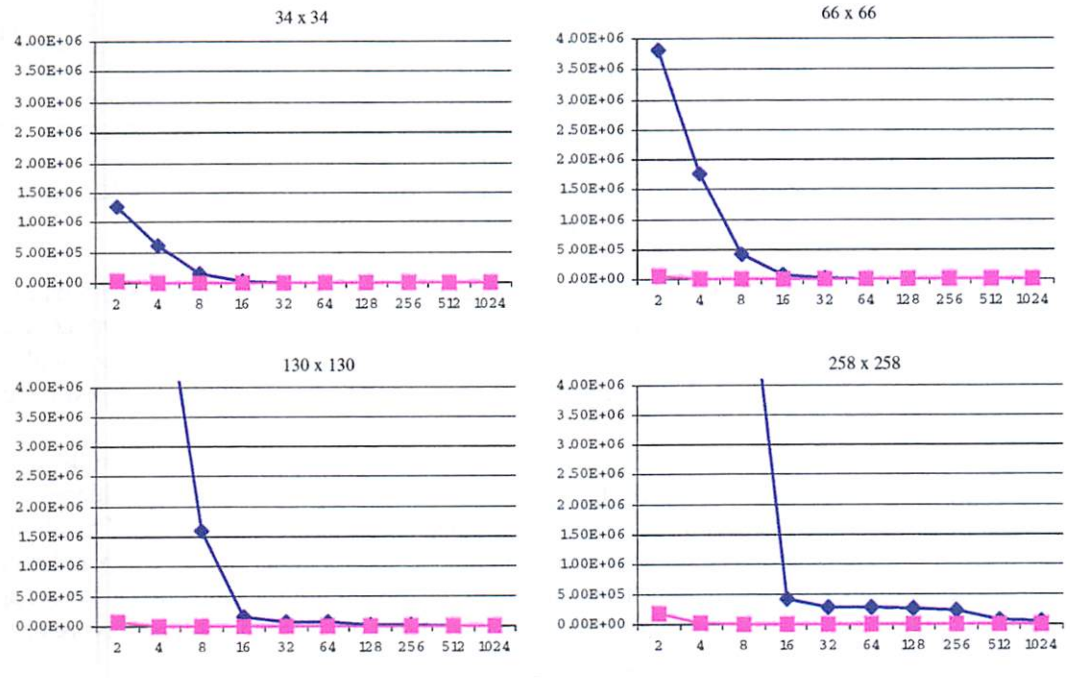
First, synonyms can be allocated in very coarse granularity and can cover a large number of pages. This has the same affect on the coverage as superpages in the TLB, but the major difference is that contiguous physical memory pages are not allocated. For example, in Figure 4.4, virtual address segments V1 and V2 are synonyms. The physical page mapping is not contiguous and most virtual pages in the synonym segments are not even mapped to physical space. Nevertheless the synonyms can be represented in one single SLB entry. For any given process, the consumption of SLB entries depends on the synonym usage instead of on the memory allocation. In general, increasing data set size does not create more synonyms --the synonyms simply cover larger memory areas.

Second, unlike the traditional TLB which holds translations covering every single virtual page accessed by the processor, the SLB is only responsible for a subset of these pages, i.e. the synonyms. The main address set does not use the SLB and, with some cooperation from the operating system, we expect that it will cover the best part of the whole virtual address space.

To demonstrate the better coverage of an SLB as compared to a TLB, we have run some simulation experiments. (The benchmarks and the experimental set-up are described in section 4.3.1) Figure 4.5 compares the number of TLB and SLB misses in four benchmarks as a function



(a) Radix



(b) Ocean

Figure 4.6. Total Number of Misses as a Function of the Number of Entries in a TLB and in an SLB for Radix with Different Data Set Size

of the number of entries. At the low end of these graphs, the number of TLB misses overwhelms the number of SLB misses by orders of magnitudes.

Figure 4.6 shows the SLB/TLB miss comparison for two Splash benchmarks, Radix and Ocean, with 4 different data set sizes. We see from these graphs that the miss curve for the TLB takes off as the data set size increases. An SLB of size 8 or bigger has negligible amount of misses,

4.2.3.2 Page Mapping Changes and TLB/SLB Shootdowns

Another important issue for V/P and V/V caches is the demapping and remapping of pages. In a V/P cache, every time a virtual address is remapped to a different physical address the TLB must be updated, which triggers TLB shootdowns in multiprocessors. In V/V caches, the cache hierarchy must sometimes be flushed.

In contrast with a TLB, not every virtual-to-physical address mapping changes need to update the SLB. Paging activities (swap-in and swap-out) as well as page migrations do not affect the SLB nor the V/V caches. The only mapping changes that affect the SLB or the V/V caches are when the *content* of a virtual page is actually changed through the re-mapping. For example when the process image is overlapped through a EXEC system call, or when a process is terminated and its virtual space is reclaimed by another process, the SLB must be updated for all addresses in the synonym set and the V/V cache must be flushed for all addresses in the main set.

Benchmarks	Remaps		SLB flush		L2-cache Flush (Blocks)	
	User	Kernel	User	Kernel	User	Kernel
Pmake	799	11311	490	10211	11008	24587
Radix	10	4471	4	4460	345	245
Rsim	6	2301	3	2275	112	738
TPC-C	18	33072	9	33028	317	943
Kaffe	923	3002	326	2925	2710	833

Table 4.1. Remaps and Flushes

Table 4.1 shows the activity associated with virtual-physical page remaps in our five benchmarks. In our simulation, we always flush either the SLB entry (synonym access) or the virtual address cache hierarchy (main address access) on a remap, even if this is not always needed. Our counts do not include the activity associated with reclaiming virtual address pages at the end of the execution.

In the table, *Remaps* is the number of virtual-physical page remaps in the execution; *SLB flush* is the number of SLB flushes. The rest of the remaps flush the cache hierarchy but not the SLB. *L2-cache Flush (Blocks)* is the total number of L2 cache blocks flushed by all these remaps.

These numbers are very small and practically negligible, when compared with the length of the trace in the simulation. The majority of the remaps are for synonyms in kernel space, which explains the very low number of L2 cache blocks that are flushed. User address remap is very rare even for Pmake, which is a multiprogramming workload.

4.2.3.3 Some Issues Specific to SLB

Memory Traps. The SLB cannot help verify/resolve memory trap conditions for virtual addresses as the traditional TLB does. Memory traps such as page faults may be triggered deep in the memory hierarchy. The performance impact of such late memory traps will be addressed in chapter 5, where it is shown that the latency of late memory traps can be largely tolerated by ILP processors.

Synonym Coherence. In the rare instance when the same data is accessed with different addresses at very fine granularity coherence may be violated. A store to an address in the synonym set may miss in the SLB and in the virtual address cache hierarchy even though its main address is allocated in the cache hierarchy. In this case, a following load accessing the same data with the main address hits in the V/V cache before the trap for the store is detected, thus bypassing the previous synonym access. Coherence is violated.

This is not an issue for processor architecture implementing sequential consistency such as the MIPS or PA-RISC architectures because memory operations are retired one after another, after they are completed. Logically, the following loads will not be issued until the previous stores are globally performed, at which point the synonym has be solved.

However for weaker memory models[25] the coherence issue is real because the stores can be retired as soon as they reach the local store buffer. We now propose some solutions in the context of weak memory models.

Since, the main address has, in general, less restrictions than its synonyms, the main address for a writable synonym is also writable. This indicates that, in the coherence violation described above, there are at least two writable virtual addresses shared within the same virtual address space. The use of synonyms within the same address space is purely optional and can be avoided by simply using the same virtual address. Most synonyms are generated and managed by the operating system kernel without the involvement of user programs. The OS kernel should not generate two writable sharings for user programs.

A more flexible solution is to allocate a quota of SLB mappings (for example 8) for each virtual address space to map writable sharings within the process. As long as the number of user writable sharings does not exceed the quota, coherence violation is prevented because suspicious stores always hit in SLB.

A general and totally flexible solution consists in checking all stores before retiring the store instruction as is done in TLBs. If a store misses in the SLB it must trap the processor to fill the SLB. Load misses in SLB are still issued to the cache hierarchy as usual. Although the SLB must now map writable main addresses, its coverage is still scalable --at least as scalable as segment registers[52]. The coverage of each SLB entry still scales well because it does not depend on the application or physical memory size.

Choosing the Main Address. Among all the virtual addresses sharing the same page, one single virtual address must be selected as the main address. The operating system should select the address with the longest lifetime as the main address, because the overhead associated with remapping a main address is much higher than remapping addresses in the synonym set. Allocating main addresses by first touch is both simple and effective because of the way the operating system fork processes. We have used first touch in our simulation experiments except for one special case, now described.

Lock Bit Optimization. The remap of physical addresses is widely used by the OS kernel to communicate with user processes and to optimize message passing in general. One common scenario is that the kernel prepares a page on behalf of a user process in kernel space and then remaps the page to user space. The kernel address is then freed and reused by other pages. The current SLB scheme may generate excessive cache flushing for this very common case, given that the kernel address is selected as the main address since it is first touched.

We propose an optimization for remap-based message-passing in general, especially for kernel buffer handling. Suppose V2 is a receiver user buffer expecting a message from V1 which is mapped to a physical page P. The current procedure is to demap V1-P and remap V2-P to pass the page. In this case V1 is the main address and its demapping is very costly.

To reduce the overhead, V2 is deemed the main address and V1 a synonym. In order to synchronize, V2 accesses must be locked out while the page is actually transferred. A lock bit is added in the page table and in the dynamic address translation hardware for the V2-P translation. This lock bit is part of the access right bits and is also copied in the V/V caches. While the lock is set, any access via V1 proceeds as usual, but an access to the page via V2 is locked out by the page table.

SLB Miss Handling. Virtually-addressed memory hierarchies considered in chapter 3 rely on unique virtual addresses which can be created by the SLB scheme. The memory hierarchy is indexed by virtual addresses in the main address set, while the synonym addresses have to be translated into main addresses by the SLB before issuing to the memory hierarchy. Although this is very rare, a virtual address missing in the SLB may eventually turn out to be a synonym. In this case, the synonym address will miss throughout the virtually-addressed memory hierarchy until at some point the virtual address has to be translated to locate the physical data. As an example, we look at the V-COMA architecture. The synonym address missing in SLB will be directed to the home node indexed by the synonym virtual address. Since the synonym address does not index the directories, it will miss in the directory look up at the home node. A NACK message sent by the home node will eventually trap the processor that issued the synonym address. The software trap handler can figure out that the virtual address is indeed a synonym. The processor will reload the SLB and reissue the virtual address which will then access the memory hierarchy with its main address.

4.3 Impact on the Miss Rate

Besides the overhead caused by SLB misses, SLB flushes, and cache flushes, one major concern associated with virtual memory hierarchies is the effect of cache addressing on miss rates.

Because the cache index function is different, conflict misses are affected. In a V/P cache accessed in parallel with a TLB or in a V/V cache accessed in parallel with an SLB the index is the virtual address. In a physical (P/P) cache the index is the physical address. Finally in a V/V cache accessed **after** an SLB, the index is the main virtual address. **From now on in this chapter we will refer to these three systems as V/P cache, P/P cache and V/V cache respectively, knowing that V/P caches and V/V caches with parallel access to SLB have the same behavior.**

The second issue is short misses in V/P caches. Short misses are just as expensive as regular misses hitting in the L2 cache. Their number grows with the cache size, contrary to conflict misses whose number tends to decrease with the cache size.

4.3.1 Methodology

Table 4.2 shows the benchmarks used in the simulations. The column titled *Instructions* shows the total number of instructions simulated, and the column titled *Loads/Stores* indicates the total number of memory accesses in our traces. These benchmarks represent different categories of applications and include a JAVA virtual machine, an OLTP commercial workload, a computer architecture simulation, a multiprogramming workload, and a compute intensive splash benchmark

Benchmarks	Instructions	Loads/ Stores	Description
Pmake	1427M	570M	4 way parallel Makes for Modified Andrew Benchmark, many small, short-lived processes that make heavy use of OS services
Radix	1212M	269M	Splash benchmark, with parameters -n2097152 -r2048 -m4194304
Rsim	4771M	2434M	Instruction driven simulator rsim[55] runs radix, sequential consistency with speculation, simulate uniprocessor for 200k cycles
TPC-C	34422M	15695M	OLTP benchmark TPC-C runs on Postgres95 database system, 4 warehouses, scale 100 times, 46 transactions mixed
Kaffe	13698M	5368M	JAVA interpreter Kaffe runs a raytracer written in java

Table 4.2. Benchmarks

We use trace-driven simulation. Traces are collected by running SimOS[37], a complete machine simulator developed at Stanford University. The simulated SGI workstation for collecting traces has 64MB of main memory, 64KB of instruction cache, 64KB of data cache, and a 1MB L2 cache. It runs the Irix 5.3 operating system. The page size is 4 KB. All user and kernel memory accesses are recorded in the trace.

With the traces, we drive the simulator of a single-processor system with a 2-level cache hierarchy. The L2 cache is fixed with a size of 1MB and is 2-way set associative. L1 cache size varies from 8KB to 1MB, and its organization is direct mapped or 4 way set associative. For V/P caches, the L2 cache is physically indexed with backpointers to L1. For V/V caches, both L1 and L2 are virtually indexed and tagged, and a 16-entry SLB is accessed *before* the L1 cache. The cache block size is 64 bytes for all caches. The LRU policy is used for all replacement.To be conservative, we always flush either the V/V cache (main address access) or the SLB (synonym access) on every page remapping, even if some of these flushes could be avoided.

In order to support the SLB in the simulation, we dynamically detect and construct sharings among virtual addresses. We maintain a page table, a reverse page table, and a segment table in the simulation. Every memory access is first checked with these tables in an efficient way before it is sent to the trace driven modules. Any new mapping or change of mapping between virtual and physical addresses triggers an update of the table structures, where we aggressively construct and merge virtual address segments and sharings among the segments. To apply the lock-bit optimization in the SLB scheme, we identify kernel pages that should have a main address in user space in a preliminary run of the trace.

4.3.2 Synonym Alternation

We first look at the characteristic of synonyms in our benchmarks. We define and detect a synonym alternation as an instance of using a different virtual address in two consecutive accesses to the same physical page. .Synonym alternation characterizes the dynamic interleaving of synonym accesses, which affects cache miss rate.

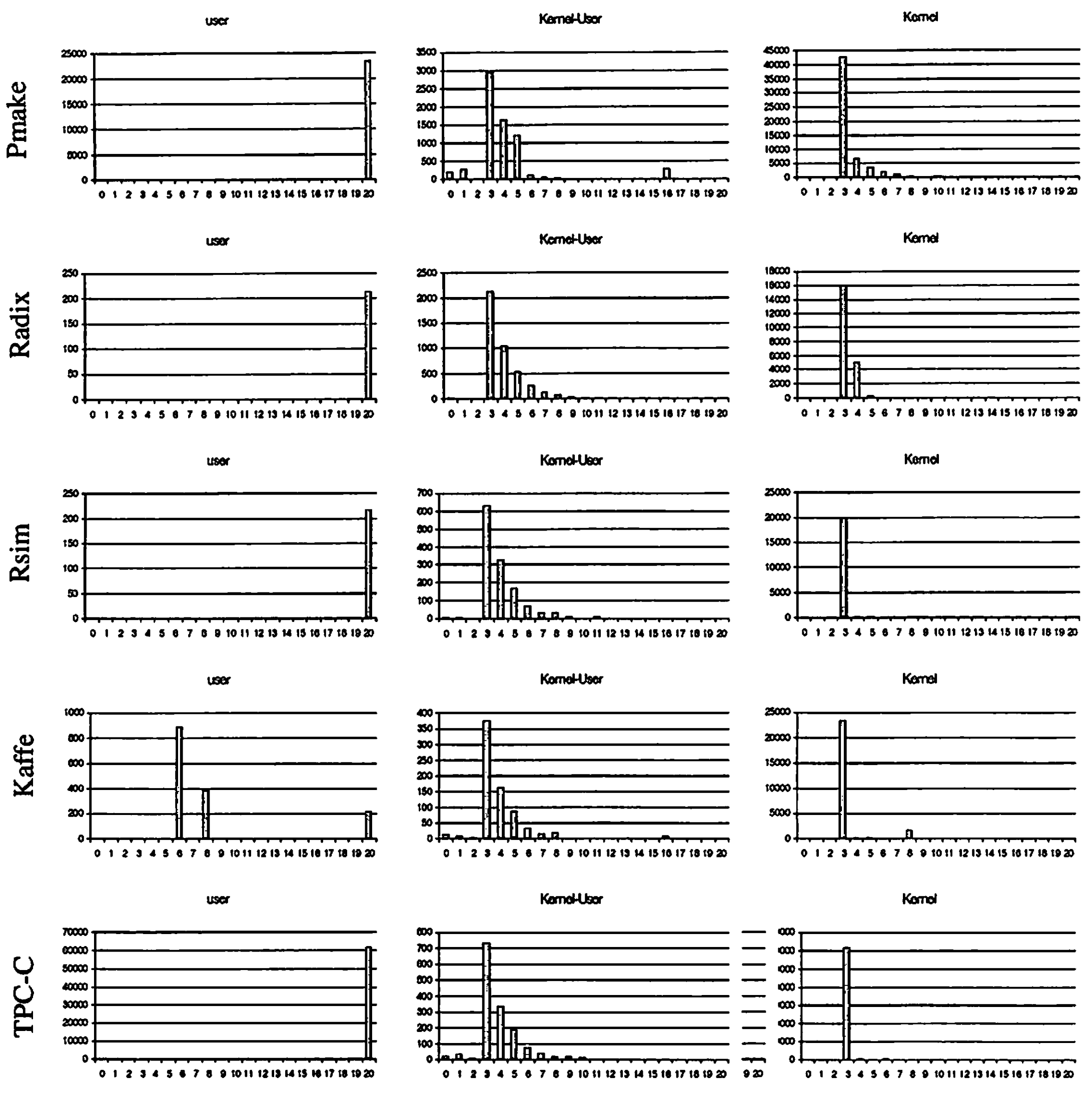


Figure 4.7. Histograms of the Degree of Proximity of Synonym Alternations

For every synonym alternation, we define the degree of proximity between the two synonyms as the number of identical least significant bits in the two virtual page numbers. In the simulation, we collect a histogram of the degree of proximity between the two synonyms in each alternation.

To interpret these results keep in mind that the least significant bits of the virtual page number are used to index a V/P cache; so, if the V/P cache index uses *n* bits of the virtual page address, then all synonym alternations with a degree of proximity less than *n* create short misses

For each benchmark, we separate the synonym alternations into three categories. *User* and *Kernel* count the cases that the two synonyms are both within user or kernel domain respectively. *Kernel-User* is for the case when the two synonyms are in different address spaces. Figure 4.7 shows the resulting histograms. The Y-axis is the number of synonym alternations, and the X-axis is the degree of proximity between the two synonyms.

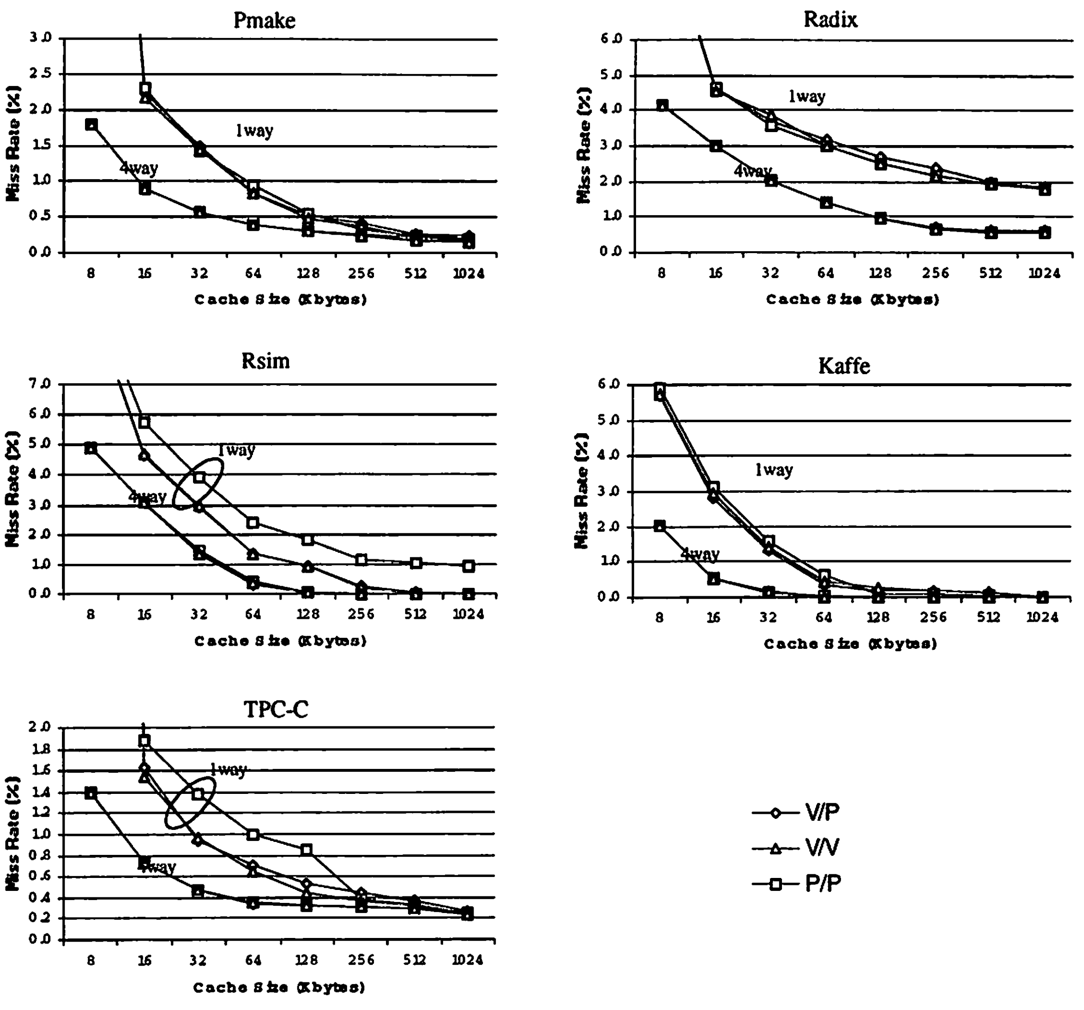


Figure 4.8. Absolute Miss Rate of L1 caches

For 32-bit virtual addresses (without counting the PID), and a page size of 4 KB, the degree of proximity ranges between 0 and 20. The figure illustrates both the total number and the distribution of synonym alternations. Except for Kaffe, user synonyms all have the same virtual addresses and index into the same cache set in any V/P cache. Unlike the other benchmarks, Kaffe contains system calls making use of the memory mapping facilities provided by the kernel.

Unlike user synonyms kernel-user and kernel synonyms may generate a large number of short misses in V/P caches. Most kernel-user synonyms pass information across the kernel-user boundary. If every block in the page is touched by both kernel and user, one synonym alternation may cause one short miss per block in the page. Within the kernel however, the reason for an alternation is not to transmit the content of a page but to share it. Each synonym usually addresses different data block at each alternation and synonyms are interleaved in fine granularity. Thus each kernel alternation creates fewer short misses as compared to kernel-user alternations.

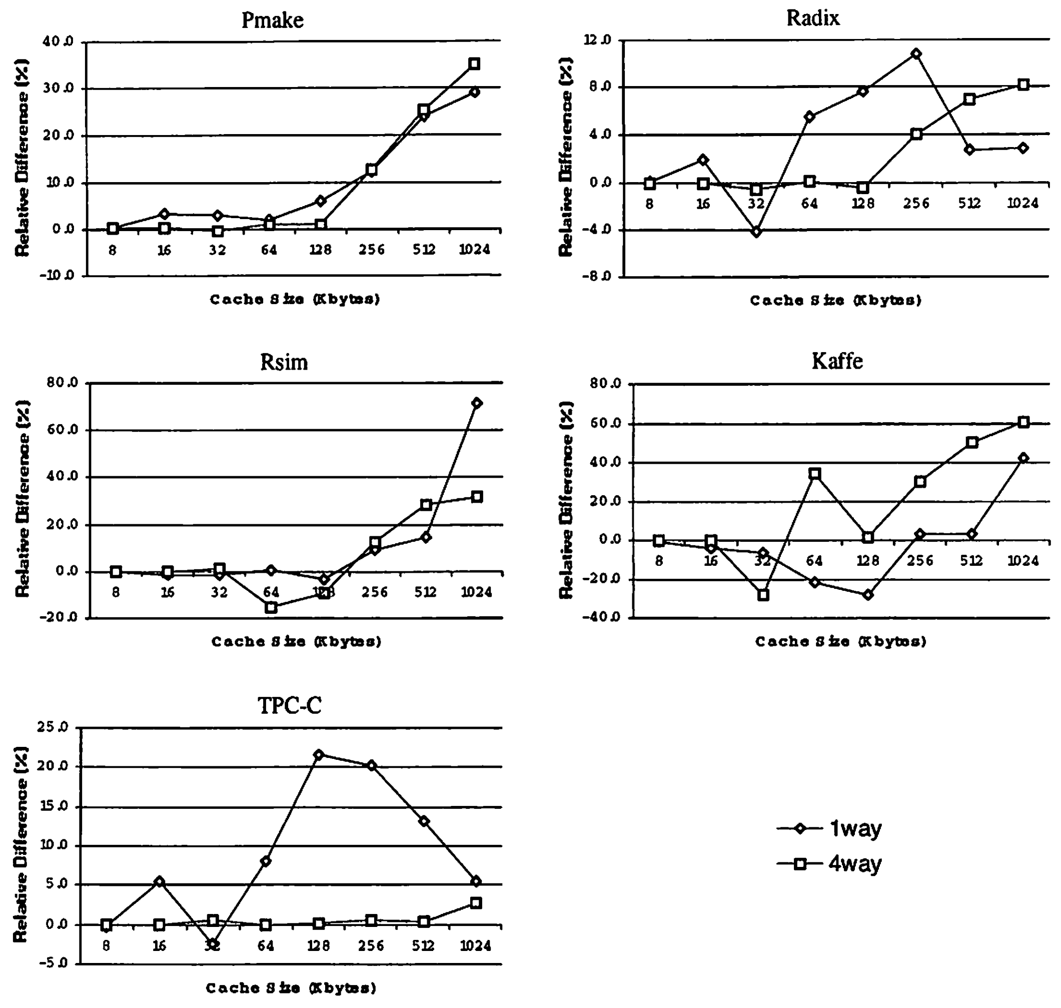


Figure 4.9. Relative Miss Rate Difference Between V/P and V/V L1 Caches

4.3.3 Overall Miss Rates

Figure 4.8 shows the miss rates, including kernel and user, for direct-mapped, and 4-way set associative L1 caches. All curves show the same trends. The index to the cache only affects conflict misses, which are significant in a direct mapped cache.

A very big gap exists between P/P caches on one hand and V/P or V/V caches on the other in the case of direct mapped caches for RSIM and TPC-C benchmark. This gap is clearly due to conflict misses. A closer look into the RSIM trace exposes that one physical page for the stack was aligned with one data page at a 1MB boundary. This suggests that for low associativity, V/P or V/V caches are more robust than physical address caches. Although the operating system can select the virtual-to-physical page mapping to minimize cache conflicts, its effectiveness at doing so is questionable, whereas virtual-address indexing takes full advantage of the spatial and temporal locality naturally exhibited by most programs[51].

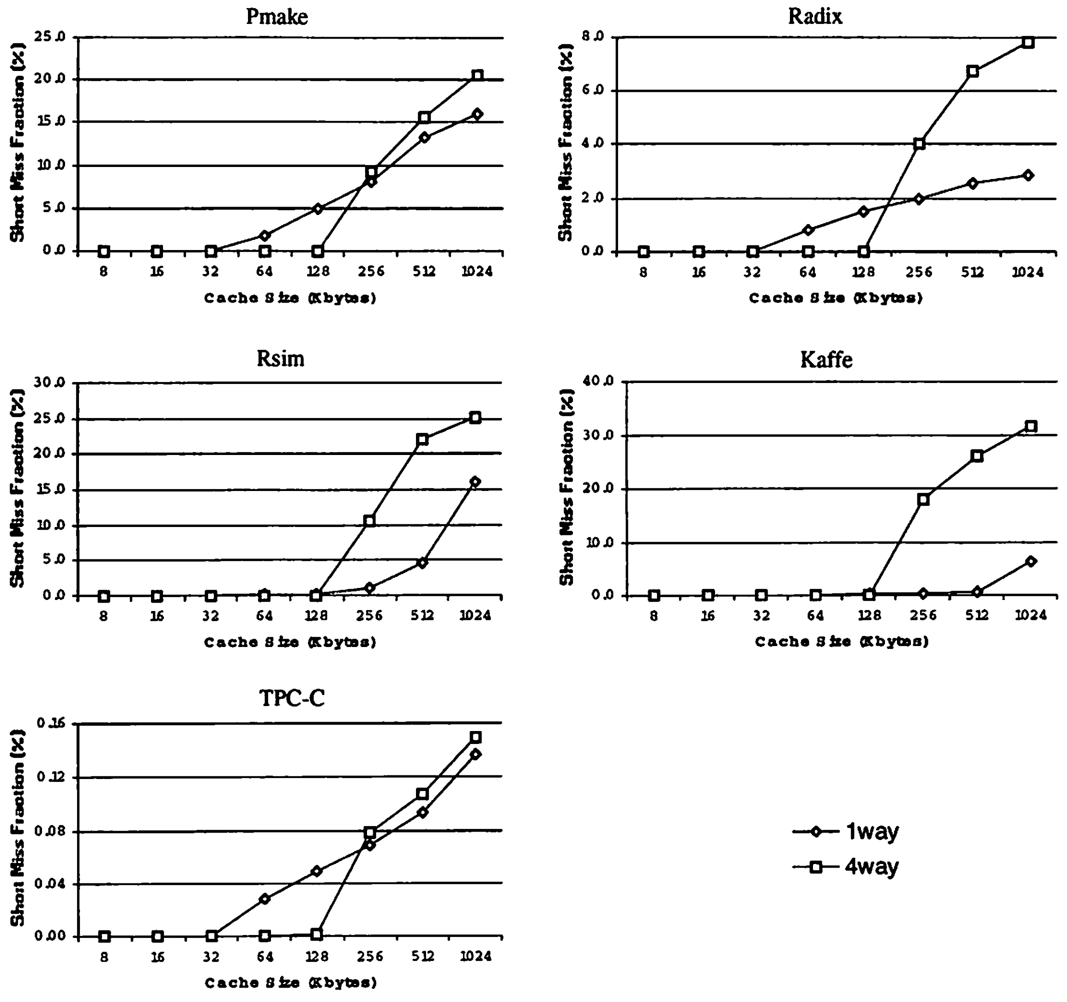


Figure 4.10. Fraction of Short Misses in V/P L1 Caches

The relative differences between the miss rates of V/V and V/P caches are displayed in Figure 4.9. They are computed as the number of V/P misses minus the number of V/V misses divided by the number of V/V misses. As the cache size increases, the impact of cache indexing becomes relatively more significant. The miss rate of the V/P cache becomes much worse than the V/V cache (up to 70% worse in RSIM). A significant part of this difference is due to short misses.

4.3.4 Short Misses

V/P caches are affected by short misses. Figure 4.10 shows the fraction of short misses in all the benchmarks, i.e. the number of short misses divided by the total number of misses for V/P caches. Again we show results for direct-mapped and four-way set-associative caches. When the cache size increases, more synonyms are indexed into different sets (see Figure 4.7) and tend to stay longer in the cache leading to an increase in short misses; at the same time, the number of capacity misses decreases sharply. We observe a sharp increase of the fraction of short misses for all benchmarks as soon as the cache size reaches 128KB. Actually if we compare the short miss rates with the relative miss rate difference in Figure 4.9, we can see that the short misses dominate the relative difference for 4 way set associative caches, especially for large cache sizes.

4.4 Summary

Moving the virtual-to-physical address translation down the memory hierarchy presents some technical challenges, among which the synonym problem is one of the major two problems. One physical address can be mapped to different virtual addresses (synonyms) which may create inconsistencies in the virtual address cache. The other major problem that of late memory traps also has many solutions, elaborated upon in next chapter.

In this chapter, we have studied the usage of the synonyms and proposed a new, simple, and effective solution to the synonym problem. In this solution, one main virtual address is used to name each virtual page uniquely in the processor and in the memory hierarchy and a Synonym Lookaside Buffer (SLB) translates synonyms into main addresses dynamically. We have given reasons why a very small SLB is effective and we have shown performance data to back this claim up. Using actual applications from different domains, we have shown that the purging and flushing of virtual-address caches is very limited in practice and has insignificant impact on performance.

Using contemporary workloads running on a real operating system (SGI Irix 5.3), we also evaluated the miss rate of virtually-addressed caches. We have presented extensive performance results comparing the miss rate behavior of physical and virtual address caches. It appears that virtual-address caches have better miss rates than physical caches and the solution using a small SLB in front of the caches avoids short misses in larger caches, while safeguarding the benefits of temporal and spatial locality in the virtual space.

Chapter 5

HANDLING MEMORY TRAPS IN ILP PROCESSORS

Traditionally, recoverable exceptions such as page faults can only be detected by the access control mechanism in the TLB which is very close to the processor pipeline. Virtually-addressed memory hierarchies require to take traps from the memory hierarchy. In this chapter, we analyze the trap behaviors of ILP processors, especially the impact of taking memory traps late. We propose new techniques to tolerate late memory traps for ILP processors.

5.1 Trapping From Memory Hierarchy

Although the semantics of memory instructions is very clear and uniform from the view-point of user programs, their implementation varies greatly. A load instruction or a store instruction issued by a program may trigger the swapping of a page from disk, an I/O operation, or a message transfer through a network. These hardware extensions are implemented by software traps on memory instructions.

Memory traps have to be generated for software to handle page faults and TLB misses. In addition to the virtual memory implementations, there are increasing demands for taking efficient memory traps, especially from the memory hierarchy. We list some examples of research ideas other than virtual memory implementation that are based on memory traps.

One important application is software and hybrid DSMs. Virtual shared memory systems[48] still rely on the traditional virtual memory access control mechanisms to generate memory traps to emulate shared memory[2][48]. Chaiken and Agarwal[13] explored a spectrum of software-extended protocols, in which traps are taken from the local directory when the directory runs out of space. Hill et al.[38] proposed cooperative shared memory which can be seen as a software-extended protocol with a single hardware pointer. Grahn and Stenstrom's software-only DSM[31] emulates directory activity in software as well. Multigrain shared memory systems[93] can be constructed by memory traps combined with a hardware coherence protocol. In Blizzard-E[71] an ECC error may trap the processor from deep in the memory system. Moga et al. evaluated a software controlled COMA (SC-COMA) where the memory controller may trap the processor when an access misses in the local node.

More generally, informing memory operations[39] have been proposed to provide feedback to software by trapping on L1 cache misses. These operations can easily be extended to any level of the memory hierarchy provided processors can take late memory traps. Finally, in [78], the TLB reach is expanded by constructing coarse grain superpages from non-contiguous physical pages, and by remapping them to "real" physical memory. To achieve this a page fault can be detected in the memory controller even though a memory access has passed the processor TLB.

These examples show that efficient memory traps from the memory hierarchy are highly desirable to promote interactions between software and hardware for memory access functions.

Without the "safety net" provided by the traditional TLB attached to the first-level cache, the processor cannot decide whether a memory access will trigger an exception at the time when it is issued to memory because the exception condition depends on the state of the data somewhere deep in the memory hierarchy. The only certainty is that the memory trap must be decided before the access is globally performed[25]. The memory consistency model must also be adhered to in multiprocessors.

5.2 Memory Traps in ILP Processors

5.2.1 Traps and Branches

Unlike external interrupts which are asynchronous to the program execution, traps are synchronous and associated with instructions. Like branches, traps change the instruction flow of programs. Intuitively, a trap acts as an implicit procedure call. It stops the current instruction flow and jumps to a predefined memory location, from which a software handler executes and finally resumes the program execution from the point in the program where the trap was taken.

Traps and branches have different behaviors although both of them interrupt the normal instruction flows. Branches can only happen on branch instructions, while traps may occur on almost every kind of instructions. If we are only interested in memory traps, memory instructions typically contribute a very large fraction of the instructions of the whole program execution. Compared to branch instructions where the branch may or may not be taken with reasonable probabilities, traps are taken on very rare occasions. Usually each kind of trap will jump to a fixed or predefined address regardless of the instructions on which they are taken. The destination address has to be in kernel address space and shared by all user programs. Therefore, unlike branches, traps will usually switch the processor to privileged mode.

In modern ILP processors, it is very costly to interrupt the instruction flow. In order to reduce the frequency of changing the instruction flow due to mispredicted branches, there have been extensive research to improve the branch prediction rate, which is extremely important for performance given the increasing degree of parallelism within the current processors. Logically, each instruction subject to traps can either complete successfully or trigger a trap on exceptions. However, a processor always predicts that every instruction will be successfully completed. The processor state will be restored if an exception happens as for mispredicted branches.

To avoid saving the processor state for all instructions, precise interrupt[75] defines a clean and simple state for the processor to save for restart. According to [75], the three following conditions have to be satisfied for precise interrupts:

- All instructions preceding the instruction indicated by the saved program counter have been executed and have modified the process state correctly.
- All instructions following the instruction indicated by the saved program counter are unexecuted and have not modified the process state.

- The saved program counter points to the interrupted instruction. The interrupted instruction may or may not have been executed, depending on the definition of the architecture and the trap.

Because of the different behaviors of traps and branches, they are implemented differently in ILP processors. As an example, we look at the MIPS R10000[92] microprocessor.

The front end of the processor has to handle branch instructions to maintain a high bandwidth for instruction fetch. In the instruction fetch stage, the instruction addresses following the branch are predicted. In the decode stage where logical registers are renamed to physical registers and the decoded instruction is appended to the active list, a branch instruction triggers a checkpoint, i.e. the processor state is saved in a branch stack from which the state can be recovered in case the branch was mispredicted. A branch mask associated with each instruction in the instruction queues and in the execution pipelines points to the depending branches. If any one of these branches is mispredicted, the instruction is aborted. Therefore, if a branch is mispredicted, the processor state is recovered immediately and all depending instructions are aborted. Because the branch stack requires a large amount of on chip resource to save the processor state, the depth of the stack is usually limited. When all the stack entries are occupied, the instruction fetch is blocked on any branch instruction.

Traps must be implemented differently because the processor can not afford to save the state for every instruction that may have exceptions. Precise interrupts are supported by the reorder buffer scheme proposed in [75]. Instructions are retired in program order based on the active list after they are completed out of program order. An exception is taken when the instruction reaches the top of the active list, at which point all previous instructions have successfully retired. The pipeline is flushed and the state is restored by unmapping the registers in reverse order of instructions in the active list, usually at the same rate as the decode rate.

5.2.2 Memory Trap Overhead

We focus on the pure hardware overhead of taking a memory trap in ILP processors. To remove the effects of the execution of a particular trap handler, we do not execute a trap handler after the trap. Rather the trapped instruction is re-executed immediately after the trap is taken, in a way similar to the soft exceptions in the MIPS R10000[92] to roll back coherence speculations.

From the perspective of the instruction fetch unit, the cycles between fetching the trapped instruction before and after the trap are wasted because no useful instruction is fetched. We call this time the *fetch-to-fetch time*. Although this is an intuitive metrics for the trap overhead, the fetch-to-fetch time overestimates the overhead of the trap because the cycles between the two fetches are not fully utilized even if the trap is not taken. The instruction fetch unit may be stalled due to various hazard conditions, or may fetch useless instructions because of mispredicted branches. Actually, the re-execution of the faulting and following instructions can be much quicker after the trap because the pipeline has been flushed.

The hardware overhead of a trap is made of two parts. The first part, which we called the *flush time*, is the time to clean up the state of the processor. It is usually proportional to the number of instructions in the active list when the trap is taken. The second part, called the *pipeline restart time*, comes from the fact that some partially executed instructions in the active list are cancelled when the trap is taken and must be repeated after the trap. Unfortunately, it is very hard to estimate the pipeline restart time in an ILP processor as the following simplified discussion illustrates.

To simplify the discussion, we consider that 1) the flush time is zero; 2) the trap does not change the program flow of the execution other than the trap itself; 3) the execution time (from issue to complete) of every instruction is not affected by the trap, and 4) a maximum of one instruction is fetched and retired in every cycle. Figure 5.1 shows the execution of four instructions when the trap is taken (E_2) and when it is not taken (E_1).

An instruction is at the same *position* in the pipeline in two different executions if it takes the same time to complete its execution from the two positions. For example in Figure 5.1, at time t_3 in execution E_2 instruction i reaches the same position it had at time t_2 in execution E_1.

When the trap is taken, each instruction in the active list has reached a particular position. When the pipeline restarts after the trap, each instruction has to reach the position it lost due to the trap before it can start doing new work. Referring to E_1 and E_2 in Figure 5.1, we define the *repeat time* of an instruction in E_2 as the number of cycles needed in E_2 to reach the same position it had in E_1 at the time when it is fetched in E_2. The repeat time is the increased execution time due to the trap. In Figure 5.1 the repeat time of an instruction in E_2 is the time from instruction fetch to the cycle pointed to by a small arrow. The pipeline restart time is the time needed to repeat all the instructions cancelled by the trap.

In Figure 5.1, the shaded area represents the increased amount of work due to the trap; it expands downwards while shrinking as more instructions are executed in E_1 and E_2 and contributes to the pipeline restart time.

The repeat time of instructions and the pipeline restart time are very difficult to estimate in an actual execution. For example, a completed load instruction flushed by the trap may have

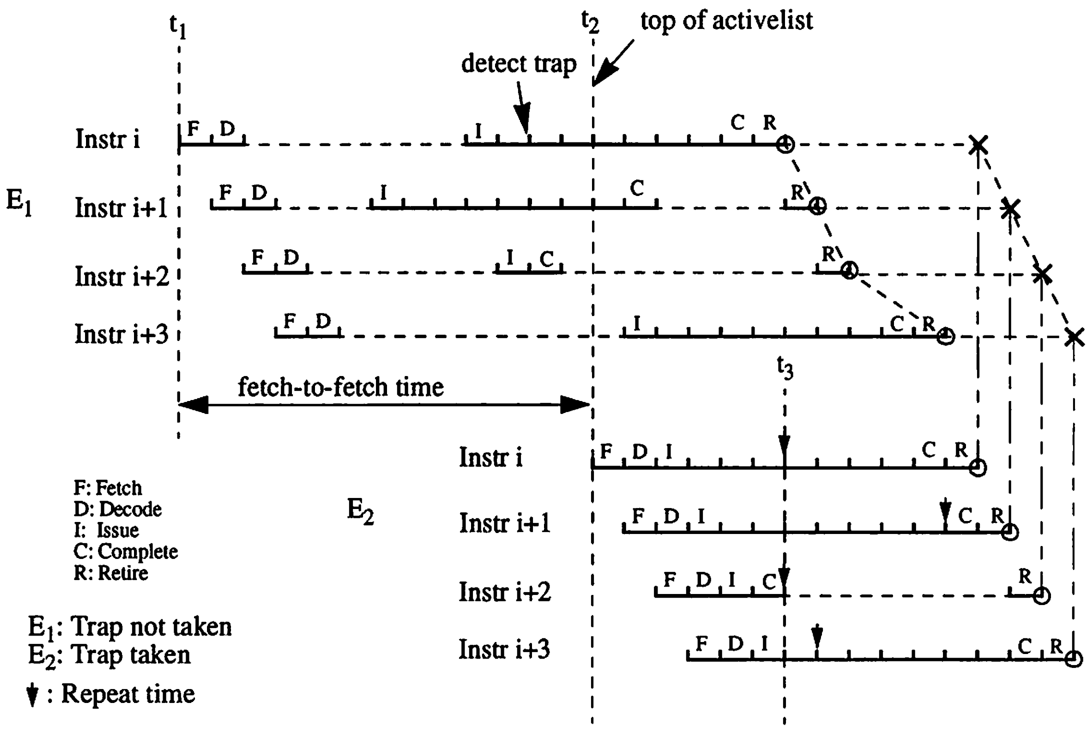


Figure 5.1. Dissection of a trap

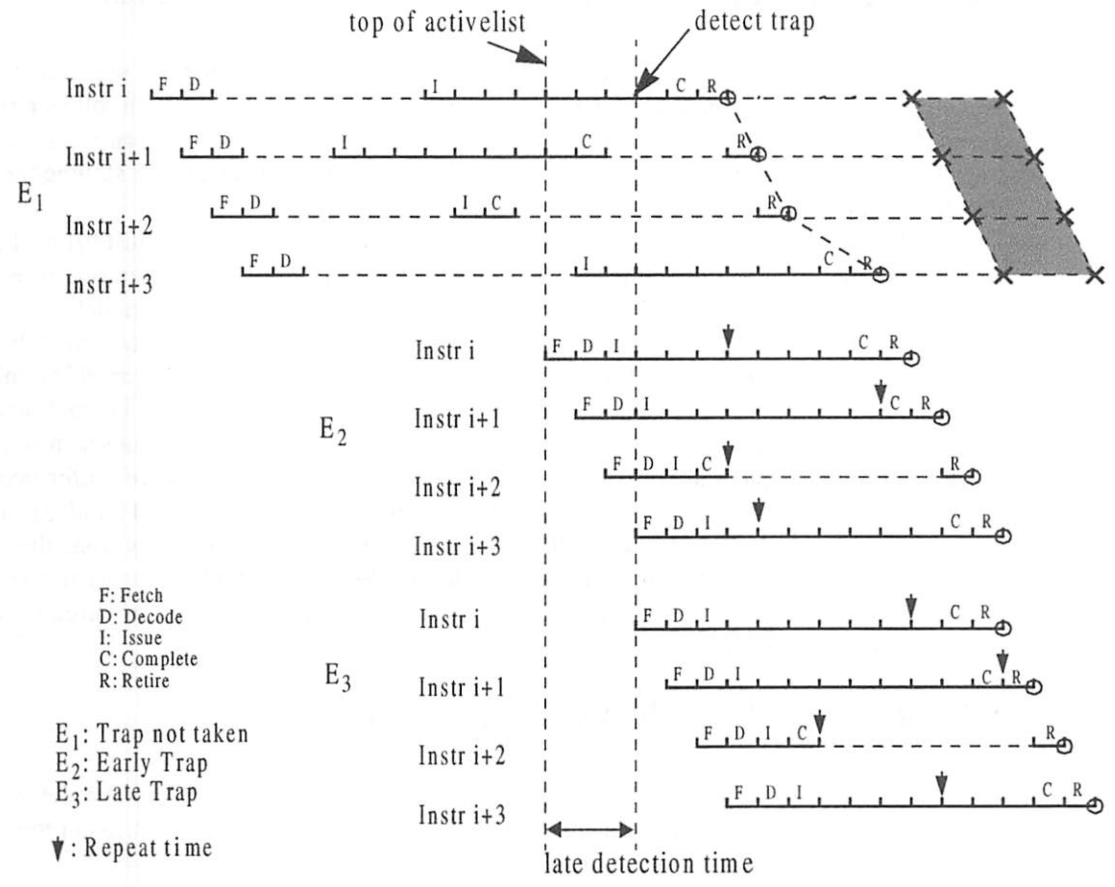


Figure 5.2. Late Detection Time

brought the data closer to the processor. This reduces the time to re-execute the instruction and therefore cuts the instruction repeat time.

5.2.3 Late Detection Time of Memory Traps

In Figure 5.1, the trap condition is detected before instruction i reaches the top of the active list. This is an *early* trap. It is possible that a trap is *late* in the sense that its condition is not resolved when the instruction reaches the top of the active list, as shown in Figure 5.2.

We define the time from the cycle when the faulting instruction reaches the top of the active list to the cycle when it is acknowledged for an exception as the *late detection time*. The processor may continue decoding instructions during the late detection time, adding to the flush time when the trap is taken. The late detection time also increases the repeat time of cancelled instructions, leading to increased pipeline restart time as shown in Figure 5.2.

5.2.4 Performance Effects of Late Traps on Non-faulting Instructions

From the perspective of exceptions, we can consider that instructions are executed speculatively until they reach the retirement point, where the program execution can rollback in case of a trap. Instructions must stay in the active list until their trap condition is verified. Late memory traps may prolong the lifetime of active list entries and increase the chance of stalling the instruction fetch unit.

A load instruction cannot be retired until its value is bound since it must fill its destination register. By that time it is guaranteed to be exception-free. Therefore, late memory exceptions do not slow down the retirement of load instructions, regardless of the memory model.

The fate of store instructions differs for different memory consistency models. A store instruction can be retired and put in a store buffer before it is globally performed[25] only if it is known to be exception-free. In a system supporting late memory traps a store instruction must stay in the active list until the exception condition is known. For sequential consistency (SC), the impact is minor. Sequential consistency does not benefit much from a store buffer because the active list stalls on a load instruction whenever the store buffer is not empty[61]. Indeed, some processors[92] do not implement store buffers and retire store instructions only after they are performed. The situation is different under weak ordering(WO)[25] and release consistency(RC)[28], which permit the overlap of memory accesses. Thus disabling the store buffer affects the performance of WO and RC implementations.

5.3 Cutting the Late Memory Trap Overhead

We propose to prefetch memory exception conditions to reduce the late detection time. Additionally, we propose to implement deferred traps instead of precise traps to cut the overhead on non faulting store instructions under RC.

5.3.1 Prefetching Memory Exception Conditions

5.3.1.1 Hardware-Controlled Prefetch

Hardware-controlled prefetch and speculative execution[29] tend to reduce the performance gap between different memory consistency models. The processor issues memory accesses as soon as possible despite memory consistency restrictions, while the hardware monitors possible consistency violations. If a prefetch violates the memory consistency model, the processor rolls back and re-executes the memory instruction.

In a plain SC implementation without prefetch, a memory access cannot be issued until all previous memory accesses are performed[25]. However, the cycles between the times when the address is ready and when the instruction reaches the top of the active list could be utilized to probe the exception conditions.

The hardware already present in ILP processors for prefetch and speculative execution can be easily enhanced to prefetch memory exception information along with the data block. The non-blocking cache must be able to handle memory exception acknowledge messages which may not contain valid data blocks. Reply messages from the lower levels of the memory hierarchy must contain access control information even though they may not include data, in which case the cache will not be refilled. If a memory request returns with a memory exception, the corresponding memory instruction is trapped. Although the prefetched data is non-binding, the prefetched exception condition must be bound with the memory instruction to avoid losing it since it is not stored in cache. Binding the prefetches for memory exception does not affect program correctness even if the exception condition is changed before the processor actually issues the memory access, at most leading to an unnecessary memory trap.

Take the example of the MIPS R10000[92]. As shown in Figure 2.5, the address queue contains all the decoded load/store instructions in their original program order. Whenever the L1 data cache is accessed by the external interface, all pending entries in the address queue are compared with the incoming address. A matching load entry sends the data directly to the destination register. To prefetch exception conditions, the incoming buffer of the R10000 must accept exception acknowledge messages, which are then forwarded to the address queue, bypassing the L1 cache. If an entry in the address queue matches the address, the memory exception bit is set for that instruction and the trap will be taken when the instruction is ready to be retired at the top of the active list.

5.3.1.2 Software Controlled Prefetch

Software controlled non-binding prefetch can also prefetch exception conditions for memory instructions. In this case, it is possible that the prefetch instruction prefetches the data before the instruction it prefetched for is decoded. If the memory block is prefetched normally, the data is inserted into the cache close to the processor as usual. However, if the prefetch instruction encounters an exception, instead of simply dropping the prefetch as in most current implementations[87], the prefetch must return the exception condition to the processor.

We propose to add an *exception buffer.* When a prefetch instruction returns an exception message, the information is copied into the exception buffer. The following memory instruction using the prefetched value is very likely to hit in the exception buffer and to detect the trap immediately.

The exception buffer can simply be a fully associative cache with the same granularity as the L1 cache line size. Each entry contains the memory address and the access control of the memory block. The size of the exception buffer is comparable to the number of MSHR registers. As for a traditional TLB, the exception buffer is searched in parallel with the L1 cache. On a hit, the instruction is trapped. No action is taken if the exception buffer misses.

Because the exception buffer is simply a temporary place to hold exception conditions, it is very easy to maintain. When it is full, the newly fetched exception condition simply overlays a selected victim. There is no need to maintain consistency as for a TLB because program correctness is not affected even if the exception buffer is stale -- a stale buffer at most triggers an unnecessary trap.

5.3.2 Deferred Trap for Memory Exception

One critical problem with late memory traps is the restriction on store buffering. We solve this problem by relaxing the precise interrupt requirement on store instruction induced traps. We retire a store instruction from the active list and put the store into the store buffer before the trap condition is verified. Following instructions are committed, and cannot be recovered from if the

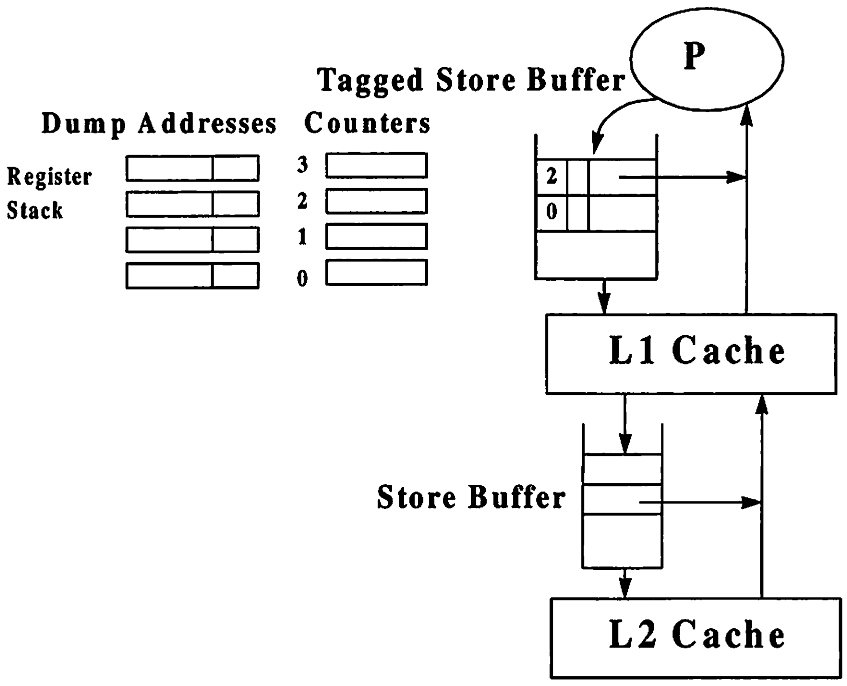


Figure 5.3. Tagged Store Buffer

store instruction triggers a memory trap some time later. The memory trap on the store instruction is not precise, but deferred in a way similar to some floating point exceptions[87].

In such a scheme, sufficient state information must be saved to resume execution. The program counter saved by the hardware may not point to the faulting instruction. The trap is taken on a subsequent instruction resumed after the trap. By the time a trap is detected on a store instruction, several other stores may be in the store buffer. They will eventually complete and may still trigger traps. It is also important to support nested traps where another trap can occur within the context of the trap handler. Since different trap levels may have different protection restrictions, the store buffer entries holding pending stores should be tagged and segregated in different trap levels for protection reasons.

We focus on the design for release consistency, but with some changes, it is also applicable to other consistency models including sequential consistency. We assume a write-back non-blocking L1 cache. The following three issues must be considered: 1) how to save and recover the state when a store buffer entry triggers a memory trap; 2) what happens if a load hits on a store buffer entry whose exception condition is not resolved; 3) how to deal with the store buffer when a trap or interrupt happen and the store buffer is not empty.

To answer these questions, we propose an aggressive design based on a *tagged store buffer* shown in Figure 5.3. In this design, when a store buffer entry triggers a memory exception, the entry is dumped to a predefined memory address. Load instructions and following instructions can use the value in the store buffer, although another option is to stall the load instruction until the store buffer entry propagates to the L1 cache and then read the value from cache. When a trap happens, the hardware does not flush the store buffer even though its entries may still cause traps.

As shown in Figure 5.3, a stack of register pairs is associated with the tagged store buffer. The depth of the stack is equal to the maximum level of nested traps that the processor can support. If the processor does not support nested traps, two register pairs are necessary. Within each level of the stack, the register pair contains a *dump address register* and a *counter register.* There are two additional fields in each store buffer entry: a trap bit indicates whether the entry encountered an exception and a tag field contains its level of traps or the index of its register pair in the stack.

When a new entry is allocated in the store buffer, the tag is set to the current level of traps. Store instructions can be coalesced only if they have the same tag value, and a load instruction returns the value in the store buffer only if it hits and the current trap level matches the tag of the store buffer entry.

Entries are retired from the store buffer out-of-order as soon as stores are successfully completed in the L1 cache or an exception is detected. The store buffer entries are compared with the address in the reply messages to the L1 cache. If an entry matches the address with no exceptions, the data is merged and copied into the L1 cache as usual.

If an exception happens, the counter register indexed by the tag of the matched store buffer entry is increased. The store buffer entry is dumped into memory at the address given by the dump address register. The dump address register contains two parts: the *base* part and the *offset* part. The offset part is increased by a fixed size which is equal to the size of the dumped entry from the store buffer. The store buffer entry that has been dumped can be freed for reuse. If the tag of the entry is equal to the current trap level, a store buffer exception signal is raised to the processor. The processor will then abort the current instruction at the top of the active list and save the processor state as if the trap was taken on that instruction. If the tag is not equal to the current trap level, no exception signal is raised immediately. Traps are generated whenever the processor returns to a trap level with a non-zero counter.

Some special instructions are provided to manipulate the store buffer and its register pairs. The handler for the store buffer trap must access the dumped entries in local private memory and globally perform those stores saved in the entries with appropriate actions, and then resume execution from the saved program counter.

For traps other than store instruction traps, the trap handlers can decide whether or not to complete the stores in the store buffer. For example, the timer interrupt handler must execute all pending store instructions before switching context. However, a software coherence handler responding to a load instruction trap might be fine tuned and may not empty the tagged store buffer.

5.4 Experimental Evaluation

5.4.1 Methodology

We have run instruction-driven simulations using the RSIM[55] simulator from Rice University, which models both processors and memory system in great detail. The processor model is based on the MIPS R10000 microprocessor. Our baseline processor architecture has a 64 entry active list, 2 integer units, 2 floating point units, 2 address calculation units, 8 branch stack entries and a 32 entry address queue. The cache line size is 64 bytes for both L1 and L2 caches. L1 cache is 16K byte write back, direct mapped with 1 cycle hit time, 8 MSHR registers and 2 request ports. L2 cache is 64K byte write back, 4 way set associative pipelined with 8 cycle hit time. It has 1 request port and 8 MSHR registers. Local memory access time is 18 cycles. Multiprocessor configurations have 16 processor nodes connected by a 4 x 4 mesh network. The network operates at half the clock rate of the main processor and is 64 bit wide. The benchmarks are shown in Table 5.1.

The purpose of the evaluations is to estimate the overhead caused by traps and, more specifically, by late memory traps. Because the executions with and without traps are not identical, it is very difficult to measure accurately the pipeline restart time as defined in section 5.2. Instead we first measure an overhead value directly related to the execution time called *the cost of a trap*. The cost of a trap is computed as the difference of the total execution times with traps and with no trap divided by the total number of traps in all processors. Remember that we do not execute a trap handler at the occurrence of a trap, we simply re-execute the code at the faulting address.

Clearly the cost of a trap is an imperfect metrics. It includes much more than the pipeline restart time plus flush time, especially in multiprocessors. It also includes secondary effects due to differences in executions such as different memory stall times or synchronization times. In the case of release consistency with precise traps it also includes the increased stall times on store instructions due to the disablement of the store buffers. Not withstanding these shortcomings, since execution time is the ultimate measure of performance, the cost of a trap is a useful metric.

Benchmark	Parameter
FFT	65536 points
RADIX	1024 radix, 512K keys, max 512K
WATER	125 molecules
MP3D	50000 particles

Table 5.1. Benchmarks

The *late detection time* is measured in the simulation with traps and indicates the increased trap time due to the lateness of the trap. The *flush time* is the number of instructions in the active list when the trap is taken divided by the rate at which instructions are flushed (in our case, 4). The *fetch-to-fetch time* is the number of cycles between the first and second times the trapped memory instruction is fetched, including the flush time.

We first show measurements on a uniprocessor simulation and then we show results for multiprocessors.

5.4.2 Uniprocessor

We simulate two basic schemes for uniprocessor: *TLB* is the conventional TLB scheme where the TLB is accessed in parallel with the L1 cache; in *L2_TLB* the TLB is accessed after a virtually indexed and tagged L1 and L2 caches. The processor is trapped on TLB misses in both schemes. The TLB maps 4K byte pages per entry, and is fully associative with LRU replacement policy. We have chosen a 32-entry TLB size for FFT, RADIX and MP3D, but we have reduced the TLB size to 16 entries forWATER in order to trigger enough traps. The processor model enforces sequential consistency (SC) as in the MIPS R10000. We compare the metrics for the plain SC implementation without prefetch and the SC implementation including hardware controlled prefetching of data and of exception conditions. In order to observe the tolerance to the lateness of memory traps, we delay the trap detection further by 15 cycles for the two base schemes. These two delayed schemes are called *TLB/15+* and *L2_TLB/*15+ in Table 5.2.

The fetch-to-fetch time is an upper bound on the overhead of the trap. It is consistently much larger than the cost of the trap measured directly from the execution times. Thus the fetch-to-fetch time is not a good estimate of the hardware overhead of a trap.

Bench-marks	Schemes	No Prefetch				Prefetch			
		late detecti on	flush	cost	fetch to fetch	late detecti on	flush	cost	fetch to fetch
FFT	*TLB*	0.01	9.78	11.58	37.73	0.00	14.91	23.75	64.06
	TLB/15+	14.89	15.99	32.29	58.06	0.72	15.58	24.30	67.32
	L2_TLB	3.96	15.99	20.60	81.95	0.08	15.48	21.80	85.61
	L2_TLB/15+	18.93	16.00	35.00	96.40	0.60	15.56	22.18	86.06
RADIX	*TLB*	0.99	15.98	18.17	69.33	1.71	15.73	28.98	47.57
	TLB/15+	15.99	16.00	33.17	84.33	8.67	15.71	37.01	58.61
	L2_TLB	5.00	16.00	21.98	80.24	3.11	15.62	30.51	51.02
	L2_TLB/15+	20.00	16.00	36.96	95.23	10.23	15.83	38.63	61.07
WATE R	*TLB*	0.02	13.35	18.73	51.11	0.01	12.62	21.60	34.33
	TLB/15+	11.75	15.80	32.76	65.29	6.94	15.54	32.16	43.85
	L2_TLB	1.88	13.80	20.90	53.14	1.09	13.35	28.46	35.21
	L2_TLB/15+	15.90	15.93	37.04	69.29	9.57	15.79	31.52	46.02
MP3D	*TLB*	0.06	12.93	18.01	54.22	0.06	12.62	19.69	42.65
	TLB/15+	12.49	15.96	33.16	69.67	10.43	15.86	29.07	55.81
	L2_TLB	2.80	14.08	21.71	58.96	1.55	13.64	22.67	45.33
	L2_TLB/15+	16.46	16.00	37.10	74.54	14.03	15.91	31.64	59.64

Table 5.2. Uniprocessor. All times are in cycles.

The cost of a trap varies widely among different benchmarks because of the different memory access behaviors. Under prefetching, the late detection time drops for all benchmarks but the extent of the drop depends on the benchmarks.

Prefetching exception conditions is effective at tolerating some or all of the trap delay. This can be seen fromTable 5.2 by comparing *TLB* with *TLB*/15+ and *L2_TLB* with *L2_TLB*/15+. In systems with no hardware prefetch, the additional 15 cycle delay of the trap is practically added to the late detection time and to the cost of the trap from *TLB* to *TLB*/15+ and from *L2_TLB* to *L2_TLB*/15+. This shows no tolerance for trap delay. The system with prefetching behaves differently, especially for FFT where most of the trap delay is hidden and the cost of the trap is practically independent of the delay of the trap. We also see some reduction in the late detection time in other benchmarks, reflected in the cost of the trap.

Looking at the results for *TLB* where most of the traps are detected early, we observe that the cost of the trap is higher under prefetch than under no prefetch. This important effect comes from the fact that the pipeline restart time is larger in a more aggressive pipeline since there may be more memory instructions cancelled by the trap. This is even more obvious in Table 5.3 for multiprocessors.

In most cases, the active list is very close to full, which is very desirable. Although the number of pending instructions increases when we prolong the trap detection time, the impact on the flush time is very limited. The flush time hovers between 13 and 16 cycles.

5.4.3 Multiprocessors

We are going to look at the overhead of traps in more detail in the context of multiprocessors. In this section, we evaluate the trap overhead for three systems called *L2_cache*, *Local* and *Home* which all rely on late memory traps. In *L2_cache* the processor is trapped on L2 cache misses to handle virtual-to-physical address translation or/and cache coherence protocol[20]. *Local* traps the processor on local node misses, which is needed by many software extended cache coherence protocols. *Home* detects the exception at the home node of the memory block, which can either be remote or local. This can happen when virtual address translations are done at the memory. Unlike *L2_cache* and *Local* which trap on every miss, *Home* generates exception on approximately 5% of the directory accesses at the home node. This 5% is much more than the frequency of traps in V-COMA, which only occur on very rare events such as page faults. In the case of release consistent (RC) memory systems, we compare the implementations with precise and deferred traps.

5.4.3.1 Trap Overhead

First, we show results for the trap overhead in the case of sequential consistency. In Table

Bench-marks	Schemes	Plain SC Implementation				Speculative SC Implementation			
		late detec-tion	flush	cost	fetch to fetch	late detec-tion	flush	cost	fetch to fetch
FFT	*L2_cache*	3.82	15.91	21.00	106.43	1.58	15.53	75.55	98.1
	Local	15.95	16.00	33.58	201.54	11.79	15.39	129.91	171.39
	Home	75.60	16.00	97.48	329.36	53.50	15.96	160.47	173.96
RADIX	*L2_cache*	4.72	15.99	23.89	121.99	3.07	15.27	45.91	80.36
	Local	12.41	16.00	35.85	140.54	11.71	15.65	60.89	92.05
	Home	109.03	16.00	173.00	266.00	111.51	15.99	125.04	179.12
WATE R	*L2_cache*	3.51	15.66	16.73	170.42	1.69	15.36	80.23	150.13
	Local	16.61	15.82	36.06	185.07	11.95	15.68	100.49	162.69
	Home	111.13	16.00	228.57	574.48	62.2	15.75	127.30	363.46
MP3D	*L2_cache*	3.20	15.09	20.21	179.10	2.07	15.05	77.42	172.30
	Local	13.07	15.67	27.49	210.88	10.36	15.54	92.04	200.47
	Home	119.39	16.00	139.80	474.90	92.93	15.83	116.65	328.94

Table 5.3. Trap Overhead in Sequential Consistency (SC)

5.3, *Plain SC Implementation* represents the scheme without prefetching of data and exception condition. In addition to comparing the plain SC implementation with the prefetch scheme for both data and exception condition, we simulate one more case that the data is prefetched speculatively but the exception condition is not fetched until the memory instruction is issued, which is shown as *Speculative SC Implementation* in Table 5.3. The same data is also appeared in Table 5.4 as *No Prefetch of Exception Condition* in order to compare against the scheme of prefetching both data and exception condition, which is shown as *Prefetch of Exception Condition* in Table 5.4.

Bench-marks	Schemes	No Prefetch of Exception Condition				Prefetch of Exception Condition			
		late detec-tion	flush	cost	fetch to fetch	late detec-tion	flush	cost	fetch to fetch
FFT	*L2_cache*	1.58	15.53	75.55	98.1	0.04	15.42	73.93	96.77
	Local	11.79	15.39	129.91	171.39	0.20	15.33	116.34	160.66
	Home	53.50	15.96	160.47	173.96	31.84	15.88	139.56	155.27
RADIX	*L2_cache*	3.07	15.27	45.91	80.36	0.82	14.87	43.45	77.75
	Local	11.71	15.65	60.89	92.05	4.21	15.23	52.63	84.26
	Home	111.51	15.99	125.04	179.12	85.49	15.94	97.40	161.85
WATE R	*L2_cache*	1.69	15.36	80.23	150.13	0.78	15.21	78.45	149.34
	Local	11.95	15.68	100.49	162.69	8.83	15.55	90.31	159.43
	Home	62.2	15.75	127.30	363.46	57.55	15.72	79.42	354.02
MP3D	*L2_cache*	2.07	15.05	77.42	172.30	0.43	14.78	76.35	170.67
	Local	10.36	15.54	92.04	200.47	6.83	15.32	89.84	197.62
	Home	92.93	15.83	116.65	328.94	78.88	15.77	102.98	343.17

Table 5.4. Prefetching Exception Condition in Sequential Consistency (SC)

As in the uniprocessor case, the fetch-to-fetch time grossly overestimates the cost of the trap. The active list is about full at the point of the trap in all cases, which means that the flush time is roughly constant and close to 16 cycles.

Table 5.3 can better explain the impact of a more sophisticated processor on the trap overhead. The cost of the trap increases significantly from *Plain SC Implementation* to *Speculative SC Implementation,* for the cases of *L2_cache* and *Local* where the restart time plus the flush time dominates the cost of the trap. In the case of *Speculative SC Implementation,* the processor efficiency is much better because of overlapped pending memory requests, which in turn increases the amount of work that is cancelled by the trap and thus the restart time.

Comparing the late detection times for the two schemes in Table 5.3, we observe that prefetching data only as the *Speculative SC Implementation* can also somewhat reduce the late detection time for most benchmarks. The memory instructions have more time between the cycle when they are ready to issue and the cycle when they reach the top of the active list. In other words, in *Speculative SC Implementation,* memory instructions are ready sooner because preceding memory instructions are completed faster.

Table 5.4 emphasizes the impact of prefetching the exception condition. Looking at the late detection times under prefetch and no prefetch, we see that the prefetching of exceptions conditions can further tolerate a large fraction of the trap delay. Because memory instructions can probe the exception condition along with the data prefetch as soon as the address is ready, the cycles that can be utilized to tolerate the lateness of trap detection is further enlarged. If we focused on the *cost of trap* of these two schemes, prefetching exception condition always give us some improvement on the total cost per trap. The improvement comes from the reduction of late detection times.

In Table 5.5 we show results for the trap overhead in a multiprocessor under RC with precise and deferred traps. The difference between the two cases is the disablement of the store buffer when precise traps are supported. Again we observe that the active list is almost full whenever a trap is taken so that the flush time is constant. Deferred traps are very effective at reducing the effect of the late traps in all cases. The reason is that, under deferred traps, a store instruction that reaches the top of the active list can retire even if it triggers a trap later on and during that time the processor does useful work. This is not the case under precise traps. We also see that the cost of a trap has been inflated dramatically under precise traps, because the cost of disabling the store buffer for *all* store instructions has been charged to the traps.

Bench-marks	Schemes	RC/Precise Trap				RC/Deferred Trap			
		late detec-tion	flush	cost	fetch to fetch	late detec-tion	flush	cost	fetch to fetch
FFT	*L2_cache*	1.47	15.56	81.63	98.26	0.16	12.79	49.13	79.05
	Local	7.43	15.42	163.72	191.38	1.18	15.08	79.06	136.81
	Home	41.24	15.96	629.53	221.51	41.57	15.93	84.47	138.51
RADIX	*L2_cache*	2.70	15.90	144.70	86.81	0.82	14.85	25.68	40.51
	Local	10.76	15.97	204.36	99.09	1.24	15.23	31.81	44.03
	Home	101.61	15.99	2538.05	193.68	16.26	15.76	45.66	68.71
WATE R	*L2_cache*	2.51	15.39	132.61	150.63	0.97	15.09	66.47	143.52
	Local	13.62	15.64	155.63	162.68	5.55	15.38	62.54	148.09
	Home	92.29	15.89	3497.87	468.81	31.28	14.77	84.12	213.69
MP3D	*L2_cache*	2.02	15.10	112.97	172.40	0.91	14.28	62.64	140.35
	Local	9.78	15.54	130.65	199.94	3.80	14.91	69.70	160.43
	Home	91.90	15.85	1370.07	359.55	44.14	15.79	151.15	218.48

Table 5.5. Precise vs. Deferred Trap in Release Consistency (RC)

If we compare the cost of traps at different trap detection locations inTable 5.3 andTable 5.4, from *L2_cache*, to *Local* to *Home*, the cost under *Plain SC Implementation* is much more sensitive to the trap detection point than under *Prefetch of Exception Condition*. Under *Plain SC Implementation* the cost varies from about 20 cycles to more than 200 cycles depending on the benchmark and on the trap detection point. However under *Prefetch of Exception Condition*, the cost of the trap is not much sensitive to the trap detection point. This is of course due to both the increased pipeline restart time and the ability to tolerate lateness of detection. For a processor with a more aggressive memory system, the overhead of traps are increased whereas the overhead due to the lateness of the trap can be largely tolerated. This trend is more pronounced if we look at Table 5.5 for RC. The precise trap requirement makes the trap detection point essential because it not only affects trapping instructions, but also impedes non-faulting store instructions.

5.4.3.2 Execution time

In this section, we present the breakdown of the execution times excluding the execution of the software handlers. Contrary to previous results presented in this paper, the execution time takes into account the frequency of traps. Because of caching effects in the memory hierarchy, the frequency of traps usually decreases when its delay increases. We follow the convention in [56] to count the stall time components, where the stall time is charged to the first instruction that cannot be retired from the active list. If all four instructions are retired in one cycle, the cycle is credited as busy. In Figure 5.4, *Flush* is the total flush time to clean up the active list when the trap is taken and *Detect* is the late detection time for the memory traps. *SC/No prefetch* represents the plain SC implementation without prefetching. *SC/prefetch* scheme prefetches data and exception condition.

The late detection time is not significant in all cases because it is only charged when taking memory traps. As we expected, it is further reduced greatly by hardware prefetch (SC) and deferred trap (RC). Without prefetch, SC restricts the issue time of memory accesses, leading to a late detection time comparable to the flushing time of the active list; when the trap detection point moves further away from the processor, the late detection time becomes dominant in the time overhead due to traps. By contrast, the late detection time contribution to the total trap cost becomes negligible when prefetch is added.

The write stall time is very small in RC executions with no traps, because the store instructions can be immediately retired from the active list as long as the store buffer is not full.

Except for RC with precise traps, the increase of the execution time over the no trap base scheme comes from taking traps. Other non-faulting memory instructions are not affected. On the other hand, the RC scheme with precise traps enforces restriction on retiring store instructions, leading to a big increase on write stall time. Actually, because it enforces similar restrictions on store instructions, the execution time for RC/precise interrupt is very close to the SC/prefetch implementation. As also shown in the figure, deferred trap reduces the late detection time as well because the processor retires useful instructions by the time it detects exceptions for the store buffer entries.

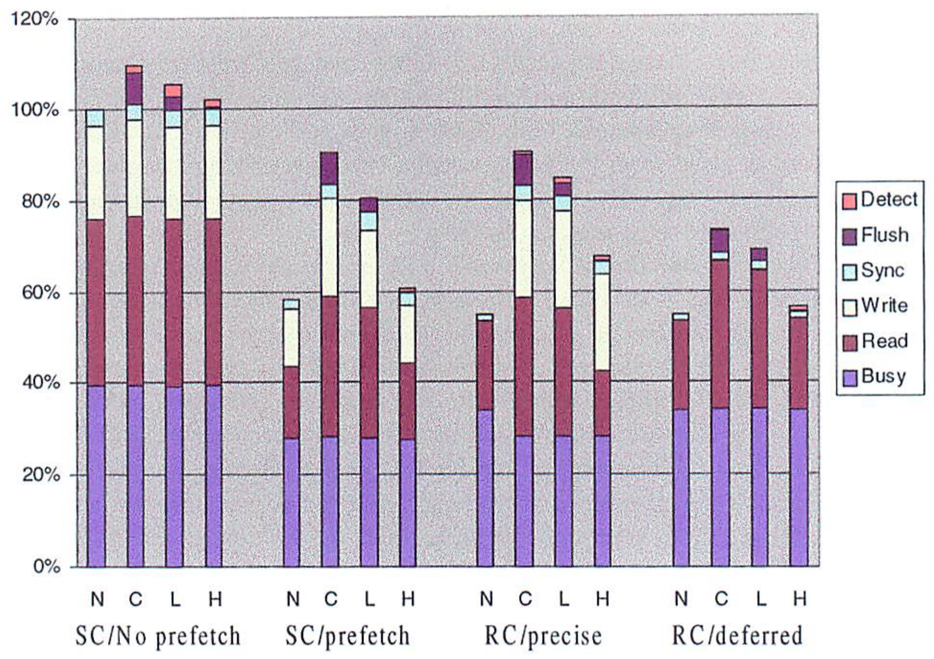
5.5 Summary

Memory traps are necessary to support the virtual memory system and many other important systems and applications. In this chapter, we have analyzed and evaluated the overhead created by traps in the context of ILP processors. More specifically, we have focused on the added overhead of late memory traps and have proposed ways to tolerate this overhead. Compared to the ideal case where the memory trap is decided as soon as the instruction reaches the top of the active list, late memory traps introduce extra overhead because 1) a store instruction cannot be retired until its trap condition is verified, and 2) memory instructions must wait at the top of the active list before the trap is detected.

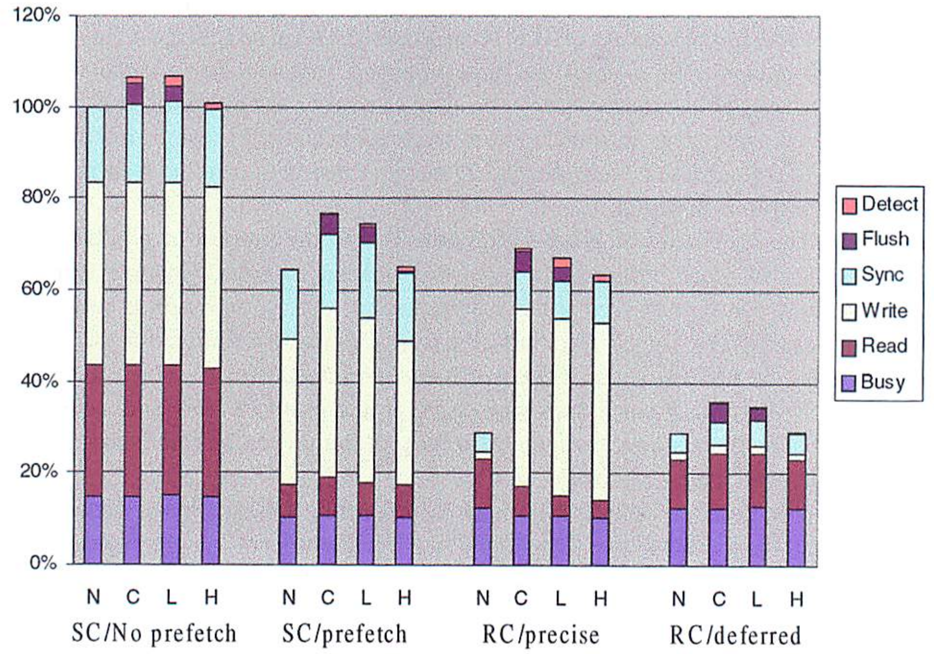
The restriction on store instructions due to the support of precise late traps destroys the advantages of store buffering in release consistent systems. However this restriction on store instructions can be removed if the requirement of precise interrupt is relaxed to that of deferred trap for stores. The tagged store buffer acts as a temporary storage for the stores whose exception conditions are not resolved, and provides enough information to resume execution after the trap.

Because ILP processors look ahead in the program flow and can start memory accesses very early in the pipeline, the lateness of trap detection can be tolerated with slight modifications of current prefetching hardware.

The deferred trap and prefetch of exception information proposed in this chapter efficiently cut the overhead due to delaying the decision time of memory exceptions. The cost of traps is high and quite unpredictable in ILP processors. As our simulation results show, when the memory system of the processor becomes more aggressive, the cost of late detection is reduced while in most cases the cost of traps increases. Systems in which these traps can be taken as late as possible will become increasingly desirable if we can take advantage of their reduced frequency due to caching effects in the memory hierarchy.

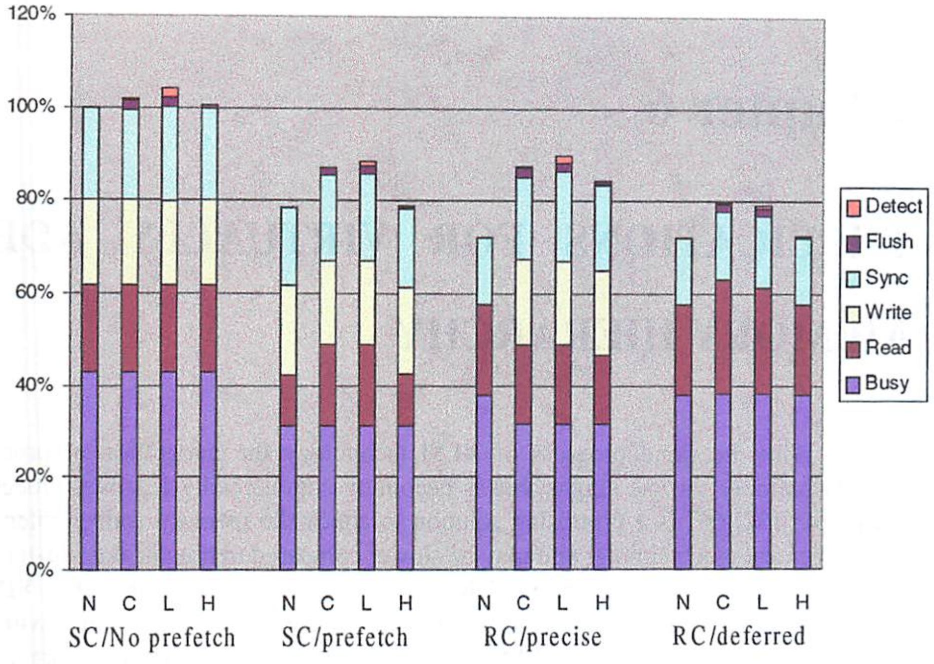


(a) FFT

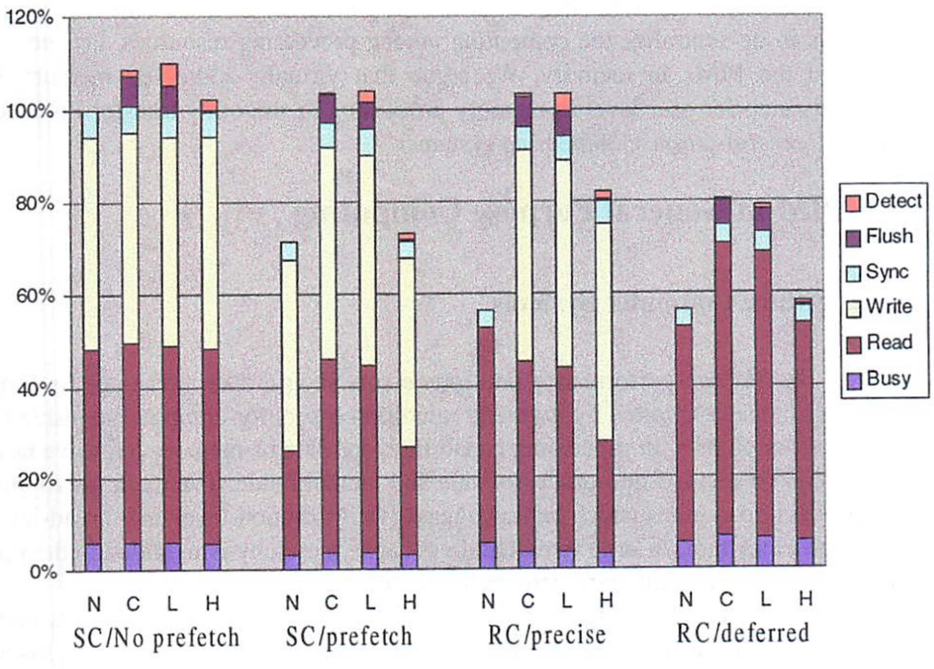


(b) RADIX

N - *No trap* C - *L2_cache* L - *Local* H - *Home*



(c) WATER



(d) MP3D

Figure 5.4. Execution Time Impact of Late Memory Traps

Chapter 6

IMPLICATIONS FOR VIRTUALLY ADDRESSED MEMORY HIERARCHY

With the rapid progress of VLSI technology the integration of processing logic and DRAM memory on one single chip is becoming commercially feasible. Processing-in-Memory (PIM) [45][57][68] is a promising solution to attack the memory wall problem. Typically, PIM processors are much simpler and may be slower compared to regular state-of-the-art microprocessors. However, since processor and DRAM are integrated on the same chip, PIM processors can exploit the huge internal bandwidth offered by the parallel memory banks within DRAM chips. Access bandwidth inside the DRAM chip is several orders of magnitude larger, and the memory access latency is reduced by up to a factor four[36], due to the elimination of off-chip communication.

The key to fight the memory wall with PIM technology to achieve high performance computing is to de-centralize the computing among processing resources, i.e. the main microprocessors and the PIMs in memory. We argue that virtually-addressed memory hierarchies are a necessary step for user level in-memory processing to smoothly transfer PIM technology into the arena of general-purpose computing systems.

6.1 PIM in General Purpose Computing

6.1.1 Future Computer Systems

Future high-performance multiprocessors should take advantage both of the instruction-level parallelism exploited by high-end microprocessors for compute-intensive tasks, and of low-cost memory chips with processing capabilities for data or memory intensive tasks. Logically, the system can be divided into processor side and memory side. The main microprocessors are high-end processing engines which exploit aggressive instruction-level and thread-level parallelism at a very high clock rate. A large virtual address cache hierarchy is included in the processor to bridge the speed and bandwidth gaps across chip boundaries.

As illustrated in Figure 6.1, the memory side contains PIM arrays and memory banks. Compared with the main processors, the PIM processors have a relatively lower clock rate, and their pipeline is much simpler. However, accesses to DRAM from the PIM processors have high bandwidth and low latency. Because of the simple processor model, the context switch time of the PIM processors is very low as well.

A challenge is to distribute the tasks among all the processing resources in order to balance the load. Contrary to a traditional multiprocessor systems, the processing units(main processor and PIMs) are specialized. Ideally, the computation should migrate to improve memory access locality.

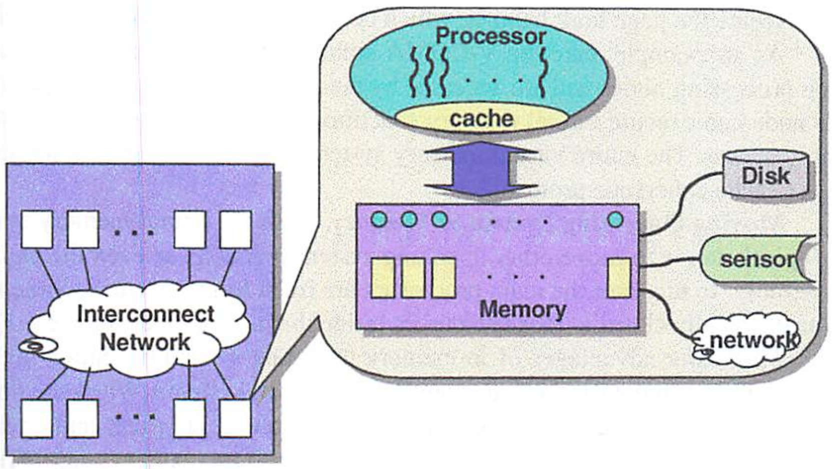


Figure 6.1. A View of Future Computer System

6.1.2 Moving Processing Towards Memory

While the main processors are still responsible for major compute-intensive tasks, resource management and memory-intensive tasks should migrate to the memory side, which is close to the data and the physical devices.

Given that the on-chip caches are virtually indexed and virtually tagged, the virtual memory system and even the entire operating system can be moved to memory executed by the PIMs. Moving the system software to memory can have significant speedup because their code and data accesses tend to be random or sequential with poor locality. Moreover, moving the system software to memory enables processing in memory at the user level. In a general purpose computing environment users should be able to program the PIMs, and the PIMs should be able to execute user programs in the virtual address space.

Now we show the logical steps involved in migrating the executions of user tasks to memory in the context of a general-purpose system.

Moving the Virtual Memory System to Memory. Once the memory hierarchy is virtually-addressed, one may consider moving the virtual memory system into memory.

In a classical system, the dynamic address translation hardware including the TLB handles the most common cases without software intervention. In a virtually-addressed memory hierarchy, we can envision as a first step that he PIM processor refills the TLB on a miss. To do that a PIM processor must be aware of the page table structures, which are also exposed to the main processors. In systems without PIMs, either microcode or a software handler triggered by a trap on the main processor must refill the TLB.

When a page fault happens, pages must be swapped in and out to satisfy the memory access. To migrate the entire virtual memory system to memory the PIMs must be able to handle page faults. To do that, the PIM processors must be aware of most of the kernel data structures, and run large amount of code. I/O operations, at least disk I/O for swapping, must be managed by the PIM processors. Actually, when the entire virtual memory system is moved to memory, the PIM processors can easily access the kernel data structures. The overall system performance can be improved because the main processor can continue to compute at the same time the PIM processor execute the page fault handler, which does not run very efficient on an ILP processor.

As an example, take the V-COMA architecture, in which the page tables are distributed among processing nodes and are accessed by the protocol processor. The protocol processor at the home node can execute virtual memory functions in case the hardware is unable to complete the memory access. The entire virtual memory system can be implemented in memory and combined with the cache coherence protocol.

Moving Operating System to Memory. With the virtual memory system in memory, the next logical step is to move other operating system functions or even the entire operating system into memory. In this case the main processors are freed from system resource management, which is done physically closer to those resources inside the memory (see Figure 6.1).

The major advantages of in-memory operating system execution are speedup, improved programmability, and easier management of PIM-based systems. When the operating system executes in memory, the main processors are dedicated to user programs and do not trap on every system call; moreover their caches are not polluted by system code and data. Much of the operating system is expected to run more efficiently in memory than in the main processor. An in-memory operating system provides a uniform environment to user tasks executed in memory and on the main processor, and PIM system programming can be done using a standard multithreading package. In-memory tasks have the same ease of access to the same system resources as the main processor tasks. Protection, security, and debugging facilities at low or no performance cost to the application programs are also provided.

Moving User Computing to Memory. The in-memory operating system can safely expose the PIM processors and memory to user programs. User level computing in memory enables portable programs for PIMs, and may change the programming model for large applications. The ability to move user and system tasks to memory has the potential to significantly relieve the well-known memory wall problem. Additionally, new programming models to explicitly expose PIMs to the user and split a computation between the main processor and memory may radically solve the problem.

Alternative Approach. The classical approach to support user-level PIM computing is to add a TLB in each PIM to translate virtual addresses, in the context of physically addressed memory hierarchies. PIM processors are treated as regular processors in a traditional multiprocessor system. User-level tasks can still execute in PIMs since virtual addresses are supported. However, as compared to virtually-addressed memory hierarchies, this approach has two drawbacks.

- The memory-intensive system software must still be executed on the main processors as well as on the PIM processors. Maintaining the consistency of the TLBs among the PIM processors and the main processors could be too expensive to make this approach feasible.

- PIM processors cannot respond to virtual addresses issued by the main processors because they can only "see" physical addresses, restricting the usage of the PIM processors. As described in the next section, user level memory functions must operate on virtual addresses. In a physical memory hierarchy, processing units exchange physical addresses. Thus the physical addresses must be reverse-translated in a reverse translation buffer (RTLB) as was done to support user-level shared memory in [63]. The effect of synonyms on such reverse translations has not really been addressed.

6.2 Emulating Memory Functions

We have shown how virtually-addressed memory hierarchies can improve the performance and scalability of the virtual memory system. As a level of abstraction, virtual memory provides each user program with the illusion of a single large memory space through cooperative hardware and software support. Virtual memory is actually a special case of what we call a *memory function*.

6.2.1 Generic Memory Functions

Application programs make some assumptions on the memory systems on which they run. These assumptions form the programmer's view of the memory access semantics. Typically, the user/system interface is the contract between user programs and their runtime environment, which includes the properties/behaviors of the memory system. The abstraction of memory semantics is usually in various layers and may be quite different for different programs. For example, a program written in the physical address space sees the memory system as a contiguous set of memory locations within a fixed address range; within that range a load returns the last value written to the same address, while the meaning is undefined outside that range. Without support for virtual memory, part of the program must be dedicated to the management of physical memory in order to provide the illusion of a big enough memory. This abstraction of the physical memory is needed by all programs; in a virtual memory system it is extracted from each individual user program and becomes common to all programs as a *memory function* of the system. A program written with virtual addresses ignores the physical limitation and expects the same access semantics throughout the entire virtual address space.

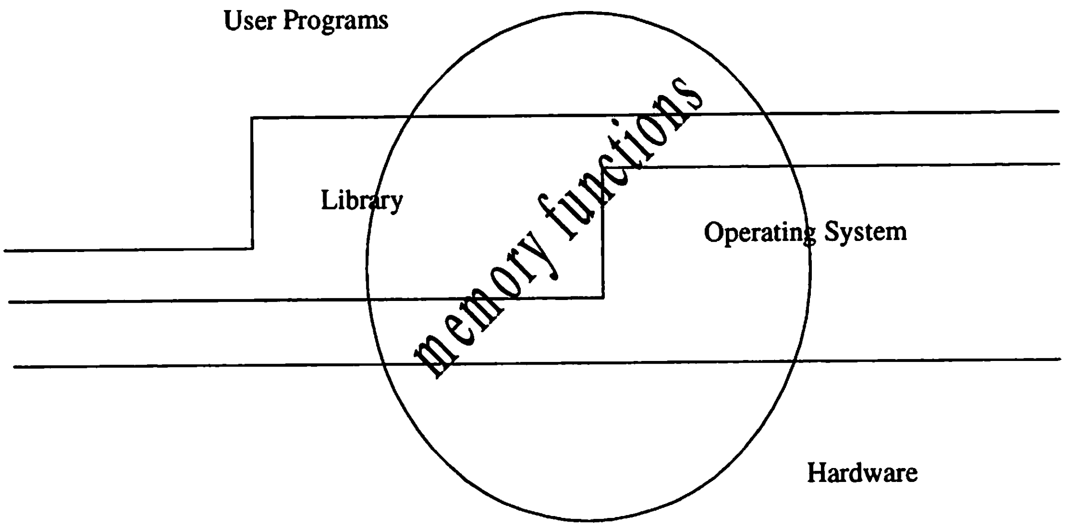


Figure 6.2. Memory Functions

A memory function is the method or action taken to maintain a particular semantics of the memory system. Memory functions can either be built in within user programs to provide another level of abstraction on top of the user/system boundary, or they can be implemented within the system as libraries, operating system functions or hardware primitives as shown in Figure 6.2.

Memory functions can be extracted from user programs and integrated within the runtime environment in which the programs run. This paradigm facilitates the portability and programmability of user applications. At the implementation level, the parallelism extracted in the memory function can be exploited by various computing agents --such as PIMs-- to improve overall system performance.

As a general paradigm, memory functions provide useful semantics to application programs. Many memory functions have already been proposed or implemented besides virtual memory. Here we list some examples. This is by no means a complete list of existing ideas. With more effective support, more innovative memory functions may emerge in the future.

Shared Memory. Instead of explicitly sending messages among processing elements, shared memory semantics lets parallel programs communicate and synchronize via memory accesses. Similar to the virtual memory abstraction, shared memory programming simplifies user programs and improves portability and programmability. Most of the time the shared-memory paradigm is supported in hardware. Virtual shared memory systems (also called software DSMs) are examples of shared memory supported by software.

Automatic Data Migration/Replication. In a distributed shared memory multiprocessor, the memory access latency is non-uniform. In order to achieve high performance, it is important to limit the number of remote memory accesses. Although user programs can be aware of the NUMA memory and manage the allocation of their data to optimize performance, the management of data allocation may also be integrated in the memory system to provide user programs with the illusion of a much simpler UMA memory access semantics. This new memory function enhances the portability and simplicity of the code and can accelerate legacy software on NUMA machines without any change to the code.

There have been many attempts to implement automatic data migration/replication for DSMs. The most notable is the COMA approach, in which automatic migration and replication are maintained in hardware. The same memory system property can be implemented in the system software layer by controlling the placement of physical data through the mapping of virtual to physical addresses, as is done in Simple COMA.

Prefetching and Smart Cache Management. The traditional cache provides support for automatic data replication controlled by hardware to reduce average access latencies. Effective prefetching can further hide memory access latencies. Better cache management, replacement strategy, or prefetch algorithm can improve latency and bandwidth to access the main memory. Complex strategies can be implemented in hardware, software, or an hybrid combination.

Garbage Collection. While some programming languages such as C specify explicitly the allocation and deallocation of memory space, programs written in JAVA or Lisp rely on garbage collection by the runtime system. To alleviate the burden on the programmer, the memory system may provide a higher level abstractions (object-oriented or functional) where memory allocation is no longer necessary.

Memory Mapped I/O and Communication. In contrast to I/O system calls, memory-mapped I/O provides user programs with a memory access semantics for I/O transfers. I/O operations are triggered implicitly through memory accesses. Since they are efficiently supported by the virtual memory system, memory-mapped file operations have been widely used. Recently, there has been extensive research on virtual memory-mapped network communications.

Informing Memory Operation. Informing memory operations[39] provide a feedback mechanism for memory accesses. Memory access conditions are explicitly monitored by trap handlers running on the processor. [39] discussed several applications of informing memory operations, such as performance monitoring, software controlled prefetching, enforcing cache coherence, and software controlled multithreading.

User-Tailored Memory System. With efficient user level support, user programs can define their own memory system semantics. This is a natural way to extract memory intensive operations to fight the memory wall problem. For example, memory forwarding[49] enables aggressive memory address layout optimizations. The Impulse project[9] pursues the same goal with a smarter memory controller.

6.2.2 Memory Function Implementation

Memory functions can either be implemented in dedicated hardware, or emulated in software. Memory functions usually execute concurrently with other parts of programs. Dedicated hardware can fully exploit the parallelism and achieve very high performance. Conventionally, tightly-coupled shared-memory multiprocessors are supported in hardware. Common cases for virtual memory system implementations are also executed in dedicated hardware. The translation lookaside buffer (TLB) translates virtual addresses into physical address and performs access right checks without any software involvement. Albeit fast, dedicated hardware is expensive and not flexible. Adding new hardware features for particular memory functions is practically impossible when using commodity components to build computer systems, as more and more of the memory system is migrated inside the microprocessor chip.

In contrast to hardware implementations, software is inexpensive. It is flexible and has more powerful functionality. Memory functions requiring very complex operations such as concurrent garbage collection are traditionally implemented in software. The software emulating memory functions can either run in the CPU along with application programs or they can be run anywhere in the memory hierarchy where there is processing support, and, in particular, within the memory chips on the PIMs.

With the trend of more aggressive ILP, higher processor clock rates and relatively longer memory access latencies, much of the processor bandwidth is unused due to various dependencies and memory access delays. This provides the opportunity to emulate more memory functions in software utilizing the unused processing bandwidth within the ILP processors. On the other hand, PIM technology expands the design space for the emulation of memory functions in software. Memory functions may execute much more efficiently in memory than in the main processors. These trends point to the programming memory functions emulated in software.

In the case of software emulation sophisticated memory functions must be extracted from application programs. These software memory functions may be common to many programs or specific to one particular program. Application programs are split into several layers which can be executed concurrently on different processing units.

When a memory function is emulated in software, one or several memory service providers logically maintain the memory system properties expected by application programs. A service provider manipulates the state of the memory units to enforce synchronization and provide the service to various parts of an application program. For a single threaded main processor, the application program usually must trap on the memory access requiring the service of a memory function and must relinquish control to the service provider for the memory functions. The service provider may also run on a different processor or as a different thread in a multi-threaded processor, and may or may not trap on memory accesses requiring the service of a memory function. The service provider may also reside in the memory hierarchy running on PIMs in parallel with the main processors.

Other parts of the application programs are generally not aware of the service provider, and do not care if it is built-in in physical memory through dedicated hardware or emulated through software running on the main processor or on PIMs in memory.

6.2.3 Support for Emulating Memory Functions

Support in Current Computer Systems. Most current commercial processors limit their hardware support for memory functions to virtual memory.

- A hardware implemented dynamic address translation mechanism is required to handle the common cases. This hardware is generally a special cache--the TLB-- holding recently accessed page table entries.
- The processor is able to take memory trap generated by the TLB, which triggers software intervention. This support is needed because complex operations such as page fault can not be completed in hardware alone. These traps are detected by the TLB when translating virtual addresses.

We have shown that the current virtual memory implementation scales poorly and is quickly becoming a performance bottleneck. In addition, it is not efficient nor extendable to support software emulations other than virtual memory system. All recoverable memory traps are generated by the traditional TLB before accesses are issued to the memory hierarchy. Other software emulations have to rely on the conventional virtual memory support as is done in virtual shared memory[48]. Although it is sometimes feasible, using the virtual memory mechanism is not efficient to support all applications. For example for software DSM, the granularity of the page table entry is too coarse to avoid false sharing. Modification of the state of data(page table) is too expensive given the high frequency of data sharing events. Some schemes propose to emulate memory functions on a separate dedicated processor in order to avoid trapping the processor from memory. Many other proposals[39][49][78] exploit various memory functions implemented by software and hardware cooperation simply by assuming that traps can be taken from the memory hierarchy even though current processors do not support it.

Emulating Memory Functions on Main Processors. In order to efficiently support the emulation of memory functions, 1) hardware on the memory access path must be added to handle the common cases of a memory function, 2) the hardware must be able to generate recoverable memory traps to the processor to request software involvement for complicated uncommon cases.

It is very unlikely that dedicated hardware will be integrated within future commercial processors to support specific memory functions as is done for virtual memory. However, the processors have to provide the hook for easy and efficient hardware and software cooperation. In other words, the processor should be able to take recoverable memory traps from the memory hierarchy so that any detection mechanism can be attached anywhere along the memory access path to trigger software handling whenever it is needed.

Virtually-addressed memory hierarchies solves the problem fundamentally. When the TLB is moved out of the processor and located in the memory hierarchy, the processor has to be able to take traps generated from deep in the memory hierarchy. The TLB and any other special hardware support can be easily inserted in the memory access path given that the virtual address caches can satisfy most memory requests. In addition, because the hardware can be accessed with virtual addresses, the support for user-level implementations is simplified. As an example, the Tempest[63] user level shared memory could be implemented much more easily in a virtually-addressed memory system, without the need for an RTLB to translate physical addresses back into virtual addresses [63].

In a virtual address cache, the cache is logical in the sense that there is no physical information involved. The dynamic address translation, as well as all other software assisted memory access implementations can be located anywhere in the memory hierarchy after the virtual address cache. The caching algorithm is general and remains unchanged despite of various memory functions emulated by software.

Emulating Memory Functions in Memory. For performance reasons, software actions to emulate memory functions should be handled as close as possible to where the actions are invoked. Memory functions that are memory-intensive should be implemented in the memory side. Memory functions is a perfect paradigm to decentralize computing among main processors and PIMs.

Take the pointer chasing problem as a simple example. User programs need to chase a pointer chain to locate some data. This piece of code does not run efficiently in a ILP processor because of the poor memory access locality and the dependencies. There are two possible approaches to splitting the user programs into compute-intensive and memory-intensive parts:

- We can encapsulate the pointer chasing code inside a procedure, which is executed on PIM processors. Therefore, PIM tasks provide memory-intensive primitives which the main processor programs can call through an interface such as the procedure call.

- The other approach is the memory function paradigm. In addition to the pointer based data structure, user programs can define another "dense and cache-friendly" data structure such as an array to represent the same data. The coherence between the two data structures can be maintained by user defined memory functions running in PIMs. Therefore, the main processor programs can access the data with good spatial and temporal locality.

The memory function approach has advantages both for program portability and performance. The service provider for the memory functions could be adaptively scheduled on the processors close to the data. There is actually no hard boundary between threads in the main processor and threads in memory. The main processor can also be used to emulate functions that are better executed in memory in case the PIMs run out of processing bandwidth. In the extreme case where dumb memory is used instead of an intelligent memory system, all tasks are executed on main processors.

In general, the application programs can be split into various layers, where different memory function service providers decentralize the computation. These service providers exploit the parallelism within applications and can take advantage of the PIM in-memory processing. The main processor programs may then have a small, clean and good-behaved data working set so that the communication between the processor and memory may be dramatically reduced.

6.3 Summary

The gap between processor and DRAM speeds keeps increasing. A radical solution to attack this memory wall problem is to split the computation and delegate the execution of memory-intensive tasks to memory.

Virtually-addressed memory hierarchies facilitates the integration of PIM processors in general-purpose computing systems. System software such as the operating system may be migrated into memory where it can run more efficiently in parallel with user programs.

Programming of mixed PIM/ILP processor systems can be done through the memory function paradigm. This paradigm is also useful in systems with no PIMs. Virtually-addressed memory hierarchies provide strong support for the implementation of memory functions, which traditionally does not exist in an architecture with physically addressed memory hierarchies. This support may come in two forms: 1) In a traditional computer system with no in-memory processors nor PIMs, the spare processor bandwidth can be better utilized to help implement a more flexible and efficient memory system. 2) In systems with PIMs, the programs emulating memory functions can be executed more efficiently on PIMs although they can also run on main processors in case a PIM is not available. In both cases, portability and programmability are enhanced.

Chapter 7

RELATED WORK

7.1 Virtual Memory

Since the first virtual memory system was introduced in the Atlas computer[27] in 1961, virtual memory has become an essential component of modern computer systems to support multi-tasking operating systems.

7.1.1 Dynamic Address Translation

Many virtual memory systems[62][87] organize the page table as a forward-mapped structure where the page table is indexed by the virtual address, possibly in multiple levels. A less common structure is the inverted page table(IPT) [40]. Since the location of an IPT entry in the table is set by the physical address but the IPT is indexed with virtual addresses the virtual address is first hashed to locate the page table entry.

Historically, a hardware finite state machine inside the memory management unit(MMU) searches the page table and refills the TLB on TLB misses. Nagle et al. [54] studied the trade-offs for software managed TLBs where a memory trap is raised on TLB misses and the software trap handler refills the TLB. Through hardware monitoring and trace driven simulations, they evaluated TLB performance on a MIPS R2000 based workstation running several different operating systems. We have shown in our study that the hardware trap overhead is much higher in ILP processors than in previous generation processors, which they assumed in their research.

Jacob and Mudge[43] looked at several virtual memory implementation choices including page table organization and TLB refill mechanism. Based on trace-driven evaluations, they concluded that the performance difference between virtual memory implementations did not really come from the choice of page table structure or MMU architecture, but rather from other implementation issues such as the trap overhead and the cache misses due to TLB miss handling.

Chen et al. [19] provided a simulation-based evaluation of TLB performance, which demonstrates the importance of the TLB mapping size on performance, which is also shown in other subsequent TLB research papers[54][80]. Although they looked at some different TLB organizations, all their designs are classified as L0-TLBs according to our characterization.

Clark and Emer[22] studied the TLB performance of the VAX 11/780 using trace-driven simulations. They showed that about 4% of execution time is due to the translation lookaside buffer. However, recent studies have shown the TLB overhead is currently much larger[54][67][79][80][65] and can even run up to 50% of the total execution time[79].

Austin and Sohi[3] evaluated the bandwidth requirement on a conventional TLB for multiple issue processors. Instead of brute force multi-ported TLBs, they compared several method to expand TLB bandwidth, including interleaved TLB, multi-level TLB, piggyback ports (which send the translation to simultaneous arriving requests), and pretranslation (which allows a single translation request to be used for multiple memory accesses). These TLB design optimizations improve the dynamic address translation bandwidth. However, they do not fundamentally solve the TLB scalability problem.

One solution to the poor performance and scalability of the traditional TLB is superpages. Talluri et al.[79][80] and Romer et al.[65] studied the use of superpages to increase the TLB coverage without enlarging the TLB. They demonstrated that superpages dramatically cut the TLB overhead by mapping physical memory in large chunks. In [79], the TLB supports two page sizes (4KB and 64 KB), and the page reservation restricts the allocation of physical memory. Their subblock TLB is similar to a subblock cache.

Romer et al.[65] advocate an on-line promotion scheme where superpages are constructed dynamically by promoting small pages to a large page. TLB misses are counted so that when the miss count reaches a threshold, a superpage is constructed by copying and reconstructing the physical memory layout. The promotion itself requires copying physical pages and updating kernel data structures and TLB shootdowns. The virtual-to-physical address mapping is too complex to satisfy the superpage requirements. The online reconstruction operations are very costly and usually increase the data set size of applications. Moreover the decision of when and how to promote superpages requires significant hardware and software efforts.

In addition to the page size, there are other factors to consider when allocating physical memory, including the page placement for NUMA memory systems and page coloring for cache friendly optimizations. These requirements sometimes conflict with each other. Moreover, changing the virtual-to-physical address mapping is very expensive in multiprocessors due to the TLB consistency overhead, which of course also limits the dynamic reconstruction of physical memory layout to utilize superpages.

The Impulse project [78] attempts to increase TLB coverage by backing up superpages with shadow physical memory. A superpage can be constructed by mapping to contiguous shadow physical pages which are then translated into non-contiguous real physical pages by the memory controller. Although this can boost performance for particular applications, it is not a general solution for superpages because shadow memory is limited and has to be managed like physical memory.

To obviate the TLB bottleneck for large applications, Saulsbury et. al. [67] proposed recency-based TLB prefetching. The temporal locality of TLB accesses is recorded in an LRU stack and stored along with the page table entries. Upon a TLB miss, nearby mappings in the LRU stack are prefetched into a TLB prefetch buffer which is accessed like the TLB. The entry is then be moved to the TLB if it hits in the following accesses.

Black et. al. [4] proposed the TLB shootdown solution to maintain TLB consistency. This algorithm does not require any special hardware support other than inter-processor interrupts. However, it requires global synchronization among all processors and all processors have to idle while a page table is being modified. Therefore, the algorithm scales very poorly in large-scale multiprocessors.

Rosenburg [66] proposed a modified TLB shootdown algorithm to improve the original algorithm. Less synchronization is required but the scheme relies on the hardware features of the IBM RP3 system.

Teller[81] addressed the scalability of maintaining TLB consistency in large scale multiprocessors. In particular, she proposed a memory-based TLB scheme in the context of UMA(Uniform Memory Access) architectures. She demonstrated that TLB consistency scales poorly in large-scale multiprocessors and that moving the TLB to memory can radically solve the problem. Her scheme is similar to SHARED-TLB in CC-NUMA and inspired the design of V-COMA.

7.1.2 Virtual Address Caches

Although virtual address caches have been the topic of many research papers, there are very few quantitative analysis of virtual address cache performance. Agarwal[1] analyzed virtual-address cache performance using traces from VAX. Wu et. al[90] evaluated different cache types including V/P and V/V caches using an IBM 370 trace. Although some of their observations are similar to the ones in this dissertation, the quantitative results are hard to compare because of the different schemes adopted for virtual address caches and of the different workloads and operating systems used in the experiments. A survey of virtual address cache issues in uniprocessors and multiprocessors has been recently published[10][11].

Synonyms have been used for various optimizations. For example, Chu [21] described a zero-copy TCP implementation using virtual-physical page remappings. Chao et. al. [15] described the porting of Mach operating system on PA-RISC, which has a virtual address cache architecture. Flushing is necessary to solve the synonym problem.

Opal[18] is a novel approach to operating system design because it runs on a single global virtual address space shared by all procedures and all data. Synonyms do not exist in an SASOS. It has become clear that the software community is clearly not ready for such a radical change.

Wang et al.[85] proposed the idea of a two-level virtual-real cache hierarchy where the TLB is after the first-level cache. We have called this system L1-TLB. They proposed to store pointers in the two caches to solve the synonym and the writeback problems and to enforce inclusion.

Among the five design options we discussed, L0-TLB, L1-TLB and L2-TLB variations have been explored by other researchers. Jacob et al.[42] proposed a software-managed address translation scheme where the hardware TLB is eliminated. A big virtually-indexed virtually-tagged second level cache drastically cuts the frequency of address translations. This scheme can be considered as a 0-entry L2-TLB.

In the VMP multiprocessor proposed by Cheriton et al.[20], dynamic address translation is also handled by trapping the main processors, which take care of the cache coherence protocol as well. Wood et al.[89] proposed an in-cache translation scheme in SPUR. Although it had a single level cache, we can categorize it as an L2-TLB scheme because there is no physically indexed cache after the address translation mechanism. As we discussed, L2-TLB is attractive in CC-NUMA or UMA architecture since the whole cache hierarchy is virtual. However, this research work was done before the advent of ILP processors.

Lynch[51] observed that physical cache performance varies in each run depending on the allocation of pages by the operating system, while on the other hand the performance of virtual-address caches is not sensitive to these implementation decisions. We have observed the same trend --i.e. the miss rates of virtual-address caches are more robust than those of physical-address caches, especially for caches with low associativity. Lynch also explored page coloring issues. His simulations indicated that the page fault rate does not noticeably increase with the number of colors. He concluded that the use of coloring did not affects paging much.

7.2 Trap and Interrupt Issues

There has been little research conducted on trap and interrupt implementations in ILP processors since Smith and Pleszkun [75] defined precise interrupt, and proposed several implementations for what they called pipelined processors. The reorder buffer scheme they proposed has been widely used in current ILP processors, and is also the base of our study on the trapping behavior in ILP processors.

Hwu and Patt [41] studied the repair mechanism for out-of-order execution machines. They looked at both branches (B-repairs) and exceptions (E-repairs), which change the instruction flows. They pointed out the different behavior of branches and exceptions and proposed different repair algorithms for them.

The lightweight trap model relying on simultaneous multithreaded architecture was recently proposed by Song and Dubois[76]. The microthread proposed in [16] is very similar to the nanothread in [76]. Zilles et. al. [94] studied lightweight exception handling for TLB refills. They have shown that the lightweight trap is more advantageous than the regular trap for TLB refills. To support lightweight traps the processor must be multithreaded. Lightweight traps cannot replace the regular memory traps that we have analyzed in this research. Lightweight traps help the main thread execution by taking care of some events that can be handled without preempting the main thread. However, complex events such as page fault require very complicated handling and most of the time preemption. For events such as TLB misses, the lightweight trap must synchronize with the main thread. The techniques proposed in this dissertation are also applicable and can be combined with lightweight traps to provide a comprehensive solution for traps in future processors.

Thekkath and Levy [82] argued that the exception mechanism was used by many applications and run-time systems in novel ways. User programs rather than the kernel should be responsible for exception handling. They proposed hardware and software support to deliver user-level exceptions to cut the overhead due to kernel context switch.

7.3 Generic Memory Functions

Traditionally, shared memory multiprocessors have been implemented by dedicated hardware. Li and Hudak[48] proposed shared virtual memory where the virtual memory support is utilized to implement shared memory system in virtual address space. This work pioneered the research on software and hybrid distributed shared memory systems, which has produced a large body of research literature and is still a very active research area.

As examples of software and hybrid shared memory implementations, Chaiken and Agarwal[13] evaluated a spectrum of software-assisted protocols. Hill et al.[38] proposed cooperative shared memory which can be considered as a software extended protocol with only one hardware pointer. Moga et al. [53] implemented a software controlled COMA (SC-COMA). Schoinas et al.[71] overviewed some fine grain access control for shared memory implementation and Reinhardt et al.[64] developed decoupled hardware support for shared memory implementations.

The SHRIMP project [5] advocated a virtual memory mapped network interface to support shared memory implementation. Similar to memory-mapped file I/O, virtual memory mapped communication maps the remote virtual or physical memory into the local virtual address space. The special network interface is designed to minimize software involvement in order to achieve high performance.

In the simple COMA architecture proposed by Saulsbury et. al[69], the virtual memory system is also involved in the cache coherence handling to allocate pages for replication. WildFire[34] connects several large SMPs to build large scale shared memory multiprocessors using an R-NUMA-like[26] protocol.

Appel and Li[2] studied virtual-memory primitives that can be utilized by user programs. They listed some applications such as concurrent garbage collection, shared virtual memory, concurrent checkpointing, persistent stores, extending addressability, data-compression paging, and heap overflow detection. Conventional virtual memory support can be used to implement these applications as demonstrated in [2]. However with better support as proposed in this dissertation, more applications can be invented given the flexibility and efficiency of the hardware/software interface. For example, in the Impulse project[78], a second-level address translation is implemented in the memory controller to improve memory behavior of user programs. Memory forwarding [49] accomplishes similar dynamic layout optimizations. It relies on memory traps to complete the memory function in software in uncommon cases. Informing memory operations[39] were proposed to provide feedback on memory behavior to software by trapping on L1 cache misses.

Burger et. al. [6] and Wulf and Mckee [91] pointed out that the increasing gap between processor and memory will lead to the so-called memory wall problem where not only the latency but also the bandwidth of memory accesses will slow down the processor. In this doomsday scenario. the processors will starve for data, especially in data-intensive applications.

Saulsbury et al. [68] described a PIM design where a very wide cache line is used to utilize the vast bandwidth within the memory chip. It includes a complete RISC processor within the memory chip with a high speed serial interconnect to communicate with other chips. The Berkeley IRAM project[57] targets in-memory vector processing. It would appear that vector processing can efficiently exploit the inherent huge bandwidth within the memory chip and, at the same time, offer the user a familiar programming style.

Kogge et al. [45] discussed programming models for PIMs to develop code for a PIM based computer. They summarized five programming models: static library model, SPMD model, modify on access model, locally shared/globally distributed model, and split execution model. In the static library model, program access to the PIM is through a library of pre-written functions. The SPMD model is simply an extension of the SPMD model for conventional multiprocessors. In the modify on access model, PIMs can execute functions which can be pushed in memory in a way similar to filters in streams. Compress and decompress modules in each level of memory hierarchies can be supported by this model. The locally shared/globally distributed model is the counterpart of message passing model for PIMs. In the split execution model, the program is broken into two concurrent executing pieces. One performs computations and the other moves data up and down in the memory hierarchy. Among the programming models they listed, modify on access and split execution model are basically special memory functions dedicated to PIMs.

The main goal of the DIVA project[36] at USC is to support emerging data-intensive applications with smart memories. A large-scale memory array is attached to a host processor. The host controls the execution in memory and executes the code that the memory array cannot execute well.

Chapter 8

CONCLUSIONS

In this dissertation, we have focussed on virtually-addressed memory hierarchies, where the virtual-to-physical address translation is moved from the processor down inside the memory hierarchy.

Virtual memory is supported by a hardware translation lookaside buffer (TLB) handling most common cases, and by a set of software functions. The traditional TLB does not scale well and is quickly becoming a performance bottleneck because: 1) The TLB is located in the critical path of memory accesses where the speed and bandwidth requirements increase with the processor clock rate and parallelism, 2) the TLB size is fixed on the processor chip and cannot be changed when building various systems, 3) TLB consistency in multiprocessors does not scale, and becomes very expensive for large systems.

Virtually addressed memory hierarchies radically solve the scalability problem of the traditional dynamic address translation mechanism and dramatically cut the overhead. When the cache memory is virtually indexed and virtually tagged, memory accesses can be satisfied without virtual-to-physical address translations. Because of the filtering effect of the virtual caches, we have shown in the dissertation that the number of address translation misses under virtually address memory hierarchies is largely reduced. In addition, virtually addressed memory hierarchies enables new designs of multiprocessor architectures. In particular, we have proposed a novel architecture called Virtual COMA(V-COMA) where the entire memory hierarchy including main memory is virtually indexed and virtually tagged. The address translation hardware is shared among all the processors and benefits from sharing and prefetching effects. Moreover, the consistency problem is eliminated because no TLB entry is replicated. Therefore, the V-COMA architecture scales very well and works even better for large scale multiprocessors.

Virtually addressed memory hierarchies have two technical challenges. One is the synonym problem, the other is the lateness of detecting memory traps.

Synonym happens when two or more virtual addresses map to the same physical address. Synonyms may generate inconsistencies in virtual-address caches and therefore are not appropriate to address the memory hierarchy in general. We propose and evaluate a new scheme called the synonym lookaside buffer (SLB) to solve the synonym problem and enable virtual-address caches. The SLB replaces the traditional TLB in the processor to handle synonyms before issuing virtual addresses to the memory hierarchy. The SLB can remain very small because its size depends on the sharing of synonyms and not on the sizes of applications or of the physical memory. We also compared the miss rate behavior of physical- and virtual-address caches. It appears that virtual-address caches exhibit better miss rates than physical-address caches and that the solution using a small SLB in front of the caches avoids short misses in larger caches while safeguarding the benefits of temporal and spatial locality in the virtual space.

Memory traps are required to implement virtual memory. Virtually-addressed memory hierarchies require processors to take late memory traps because the dynamic virtual address translation detecting exceptions is located somewhere in the memory hierarchy instead of within the processor close to the pipeline. We propose techniques to enable the ILP processors to take late memory traps. We show that, while the ILP trend in modern processors dramatically increase the cost of traps, the ability to tolerate the lateness of memory traps is also largely increased. Therefore with the techniques we proposed, the cost of traps becomes much less sensitive to where the trap is detected. Given the fact that the frequency of traps generally goes down dramatically when moving the TLB down the memory hierarchy, it becomes advantageous to take traps as deep as possible in the hierarchy rather than to take them at the level of the first-level cache, as is usually done.

We have run extensive simulations using state-of-the-art simulation tools to evaluation schemes and ideas throughout this research. Based on the type of evaluation, we have modeled both a detailed processor microarchitecture and a complete machine running a real commercial operating system. Our simulation results show that: 1) the synonym lookaside buffer scales well and efficiently solves the synonym problem for virtually addressed memory hierarchies, 2) late memory traps can be efficiently supported in modern ILP processors, and 3) virtually addressed memory hierarchies dramatically cut the overhead due to virtual memory implementation.

Virtually addressed memory hierarchies not only provide more efficient and scalable implementation for virtual memory, but also open the door to more flexible and smarter memory systems. Virtualizing the memory hierarchy is a necessary step to facilitate the integration of processing-in-memory (PIM) technology into general-purpose computing systems. Some or all operating system functions can be migrated to memory so that they can be executed more efficiently and in parallel with user programs and so that user level in-memory computing becomes possible. Virtually-addressed memory hierarchies provide better support for the general memory function paradigm so that 1) new memory functions can be developed for systems with no PIMs, and 2) applications programs can be naturally split into programs running on ILP main processors and on PIMs in memory following the memory function paradigm.

In summary, virtually addressed memory hierarchies are feasible and efficient and are very attractive for new, emerging system ideas and technologies in the context of general-purpose computing.

Bibliography

[1] Anant Agarwal, "Analysis of cache performance for operating system and multiprogramming," Kluwer Academic Publishers, Boston, 1989.

[2] Andrew Appel and Kai Li, "Virtual Memory Primitives for User Programs", In *Proceedings of the 4th Conf. on Architecture Support for Programming Languages and Operating Systems(ASPLOS)*, pages 96-107, 1991.

[3] Todd Austin and Gurindar Sohi. "High-Bandwidth Address Translation for Multiple-Issue Processors," In *Proceedings of the 22nd Annual International Symposium on Computer Architecture(ISCA)*, pages 158-167, 1996.

[4] David Black, Richard Rasgid, David Golub, Charles Hill, and Robert Baron, "Translation Lookaside Buffer Consistency: A Software Approach," In *Proceedings of the 3rd Conf. on Architecture Support for Programming Languages and Operating Systems(ASPLOS)*, pages 113-122 1989.

[5] M. Blumrich, K Li, R. Alpert, C. Dubnicki, E. Felten, and J, Sandburg, "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer," In *Proceedings of the 21st Annual International Symposium on Computer Architecture(ISCA)*, pages 142-153, 1994.

[6] Doug Burger, James Goodman, and Alain Kagi, "Memory Bandwidth Limitations of Future Microprocessors", In *Proceedings of the 22nd Annual International Symposium on Computer Architecture(ISCA)*, pages 78-89, 1996.

[7] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam, "Compiler-Directed Page Coloring for Multiprocessor," In *Proceedings of the 7th Conf. on Architecture Support for Programming Languages and Operating Systems(ASPLOS)*, Oct. 1996.

[8] H. Burkhardt III et al. "Overview of the KSR-1 Computer System," Technical Report KSR-TR-9202001, Kendall Square Research, Feb. 1992.

[9] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture(HPCA)*, January, 1999.

[10] Michel Cekleov and Michel Dubois. "Virtual-Address Caches, Part 1: Problems and Solutions in Uniprocessors", IEEE *Micro*, pages 64-71, Sep/Oct. 1997.

[11] Michel Cekleov and Michel Dubois, "Virtual-Address Caches, Part 2: Multiprocessor Issues," IEEE *Micro*, pages 69-74, Nov/Dec 1997.

[12] L. Censier and P. Feautrier. "A New Solution to Coherence Problems in Multicache Systems," *IEEE Transactions on Computers*, C-27(12):1112-1118, Dec., 1978.

[13] David Chaiken and Anant Agarwal, "Software-Extended Coherent Shared Memory: Performance and Cost", In *Proceedings of the 21st Annual International Symposium on Computer Architecture(ISCA)*, pages 314-324, 1994.

[14] Albert Chang and Mark Mergen, "801 Storage: Architecture and Programming", *ACM Transaction on Computer Systems*, Vol 6, No. 1, February 1988, Pages 28-50.

[15] Chia Chao, Milon Machey, Bart Sears, "Mach on a Virtually Addressed Cache Architecture", *Proceedings of the 1st Mach USENIX Workshop*, pages 31-51, Oct. 1991.

[16] R Chappel, J Stark, S. Kim, and Y. Patt, "Simultaneous Subordinate Microthreading (SSMT)", In *Proceedings of the 26th Annual International Symposium on Computer Architecture(ISCA)*, May 1999

[17] Alan Charlesworth, “STARFIRE: Extending the SMP Envelop”, *IEEE Micro*, pages 39-49 January/February 1998.

[18] Jeffrey Chase, Henry Levy, and Michael Feeley, “Sharing and Protection in a Single-Address-Space Operating System,” In *ACM transaction on computer systems*, pages 271-307, Nov., 1994.

[19] J. Bradley Chen and Anita Borg. “A Simulation Based Study of TLB Performance,” In *Proceedings of the 19th Annual International Symposium on Computer Architecture(ISCA)*, pages 114-123, May 1992.

[20] David Cheriton, Gert Slavenburg, and Patrick Boyle, “Software-Controlled Caches in the VMP Multiprocessor”, In *Proceedings of the 13rd Annual International Symposium on Computer Architecture(ISCA)*, pages 366-375, 1986.

[21] Hsiao-keng Jerry Chu, “Zero-Copy TCP in Solaris”, *1996 USENIX Technical Conference*, pages 253-264, January 22-26, San Diego, CA

[22] D. W. Clark and J. S. Emer, “Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement,” In *ACM Transactions on Computer Systems*, vol. 3, no. 1, February, 1985.

[23] Richard Crisp, “Direct Rambus Technology: The New Main Memory Standard”, *IEEE Micro*, pages 18-28, Nov./Dec., 1997.

[24] David Culler and Jaswinder Pal Singh with Anoop Gupta, “Parallel Computer Architecture”, Morgan Kafmann, 1999.

[25] Michel Dubois, Christoph Scheurich, and Faye Briggs, “Memory Access Buffering in Multiprocessors”, In *Proceedings of the 13rd Annual International Symposium on Computer Architecture(ISCA)*, pages 320-328, 1986.

[26] Babak Falsafi, and David Wood, “Reactive NUMA: A Design for Unifying S-COMA with CC-NUMA,” In *Proceedings of the 24th Annual International Symposium on Computer Architecture(ISCA)*, 1997.

[27] J. Fotheringham, “Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store,” *Communication of the ACM*, 4(10):435-436, Oct., 1961.

[28] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy, “Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors,” In *Proceedings of the 4th Conf. on Architecture Support for Programming Languages and Operating Systems(ASPLOS)*, pages 245-257, 1991.

[29] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy, “Two Techniques to Enhance the Performance of Memory Consistency Models”, In *Proceedings of the International Conference on Parallel Processing*, pages I355-I364, 1991

[30] J. R., Goodman, “Coherency for Multiprocessor Virtual Address Caches,” In *Proceedings of the 2nd Conf. on Architecture Support for Programming Languages and Operating Systems(ASPLOS)*, 1987.

[31] Hakan Grahn and Per Stenstrom, “Efficient Strategies for Software-Only Directory Protocols in Shared-Memory Multiprocessors”, In *Proceedings of the 22nd Annual International Symposium on Computer Architecture(ISCA)*, pages 38-47, 1995.

[32] L. Gwennap, “Design Concepts for Merced, Forecasting the Inner Workings of the Decade's Most Anticipated Processor,” pages 9-11, *Microprocessor Report*, vol. 11, no. 3, March 10, 1997.

[33] Linley Gwennap, “Alpha 21364 to Ease Memory Bottleneck,” Microprocessor Report, Oct. 26, 1998

[34] Erik Hagersten and Michael Koster, “WildFire: A Scalable Path for SMPs”, In *Proceedings of the 5th International Symposium on High Performance Computer Architecture(HPCA)*, January, 1999.

[35] Erik Hagersten, A. Landin, and S. Haridi. "DDM-A Cache-Only Memory Architecture," IEEE *Computer*, vol. 25, no. 9, pages 44-54, Sep. 1992.

[36] Mary Hall, Peter Kogge, Jeff Koller, et al, "Mapping Irregular Application to DIVA, a PIM-based Data-Intensive Architecture", in *proceedings of SC99*, Nov., 1999.

[37] Stephen A. Herrod, "Using Complete Machine Simulation to Understand Computer System Behavior", *Ph.D Thesis*, Stanford University, Feb, 1998.

[38] Mark Hill, James Larus, Steven Reinhardt, and David Wood, "Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors", *ACM Transactions on Computer Systems*, 11(4):300-318, Nov. 1993.

[39] Mark Horowitz, Margaret Martonosi, Todd Mowry, and Michael Smith, "Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors", In *Proceedings of the 23rd Annual International Symposium on Computer Architecture(ISCA)*, pages 260-270, 1996.

[40] Jerry Huck, and Jim Hays. "Architecture Support for Translation Table Management in Large Address Space Machines," In *Proceedings of the 20th Annual International Symposium on Computer Architecture(ISCA)*, pages 39-50, 1993.

[41] Wen-mei W. Hwu, Yale N. Patt, "Checkpoint Repair for Out-of-order Execution Machines," In *Proceedings of the 14th Annual International Symposium on Computer Architecture(ISCA)*, pages 18-26, 1987.

[42] Bruce Jacob and Trevor Mudge. "Software-Managed Address Translation," In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture(HPCA)*, Feb. 1997.

[43] Bruce Jacob and Trevor Mudge. "A Look at Several Memory Management Units, TLB-Refill Mechanisms, and Page Table Organizations," In *Proceedings of the 8th Conf. on Architecture Support for Programming Languages and Operating Systems(ASPLOS)*, 1998.

[44] T. Joe. "COMA-F: A Non-Hierarchical Cache Only Memory Architecture," PhD. Thesis, Stanford, 1995.

[45] Peter Kogge, Jay Brockman, Thomas Sterling, and Guang Gao, "Processing In Memory: Chips to Petaflops", In *Proceedings of the 24rd Annual International Symposium on Computer Architecture(ISCA)*, Workshop on Mixing Logic on DRAM, 1997.

[46] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. "Architecture Support for Single Address Space Operating System," In *Proceedings of the 5th Conf. on Architecture Support for Programming Languages and Operating Systems(ASPLOS)*, pages 175-186, Oct. 1992.

[47] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. "The Stanford FLASH Multiprocessor," In *Proceedings of the 21st Annual International Symposium on Computer Architecture(ISCA)*, pages 302-313, 1994.

[48] Kai Li and Paul Hudak, "Memory Coherence in Shared Virtual Memory Systems", *ACM Transaction on Computer Systems*, 7(4):321-359, Nov. 1989.

[49] C. Luk and T. Mowry, "Memory Forwarding: Enabling Aggressive Layout Optimizations by Guaranteeing the Safety of Data Relocation", In *Proceedings of the 26th Annual International Symposium on Computer Architecture(ISCA)*, May 1999.

[50] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transaction on Computers*, Vol. C-28, No. 9, Sep. 1979.

[51] William Lynch. "The Interaction of Virtual Memory and Cache Memory," *Ph.D. Thesis*, Technical Report CSL-TR-93-587, Stanford University, 1993.

[52] C. May, E. Silha, R. Simpson, and H. Warren, Eds. "The PowerPC Architecture: A Specification for a New Family of RISC Processors," Morgan Kaufmann Publishers, San Francisco CA, 1994.

[53] Adrian Moga, Alain Gefflaut, and Michel Dubois, "Hardware vs. Software Implementation of COMA", In *Proceedings of the 1997 Int'l Conference on Parallel Processing*, pages 248-256, August 1997.

[54] David Nagle, Richard Uhlig, Tim Stanley, Stuart Sechrest, Trevor Mudge, and Richard Brown. "Design Tradeoffs for Software-Managed TLBs," In *Proceedings of the 20th Annual International Symposium on Computer Architecture(ISCA)*, pages 27-38, 1993.

[55] Vijay Pai, Parthasarathy Ranganathan, Sarita Adve, "RSIM Reference Manual", Technical Report 9705, Department of Electrical and Computer Engineering, Rice University, Aug, 1997

[56] Vijay Pai, Parthasarathy Ranganathan, Sarita Adve, and Tracy Harton, "An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors", In *Proceedings of the 7th Conf. on Architecture Support for Programming Languages and Operating Systems(ASPLOS)*, pages 12-23, Oct. 1996.

[57] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick, "A Case for Intelligent RAM: IRAM", *IEEE Micro*, April, 1997.

[58] Xiaogang Qiu and Michel Dubois, "Options for Dynamic Address Translation for COMAs", In *Proceedings of the 25th Annual International Symposium on Computer Architecture(ISCA)*, pages 214-225, 1998.

[59] Xiaogang Qiu and Michel Dubois, "Tolerating Late Memory Traps for ILP Processors", In *Proceedings of the 26th Annual International Symposium on Computer Architecture(ISCA)*, pages 76-87, 1999.

[60] Xiaogang Qiu and Michel Dubois, "The Case for Virtually-Addressed Memory Hierarchies", Technical report CENG 00-03, Department of Electrical Engineering - Systems, University of Southern California.

[61] Parthasarathy Ranganathan, Vijay Pai, and Sarita Adve, "Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models", In *Proceedings of 9th Annual ACM Symposium on Parallel Algorithms and Architectures(SPAA)*, pages 199-210, 1997.

[62] SGI, Inc. "MIPS R10000 Microprocessor User's Manual," Version 2.0.

[63] Steven Reinhardt, James Larus, and David Wood, "Tempest and Typhoon: User-Level Shared Memory", In *Proceedings of the 21st Annual International Symposium on Computer Architecture(ISCA)*, page 325-336, 1994.

[64] Steven Reinhardt, Robert Pfile, and David Wood, "Decoupled Hardware Support for Distributed Shared Memory ", In *Proceedings of the 23rd Annual International Symposium on Computer Architecture(ISCA)*, page 34-43, 1996.

[65] Theodore H. Romer, Wayne H. Ohlrich, and Anna R. Karlin. "Reducing TLB and Memory Overhead using Online Promotion," In *Proceedings of the 22nd Annual International Symposium on Computer Architecture(ISCA)*, page 176-187, 1995.

[66] Bryan Rosenburg, "Low-Synchronization Traslation Lookaside Buffer Consistency in Large Scale Shared-Memory Multiprocessors," In *Proceedings of the 13th Symposium on Operating System Principles*, 1989.

[67] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström, " Recency-based TLB Preloading", In *Proceedings of the 27th Annual International Symposium on Computer Architecture(ISCA)*, 2000.

[68] Ashley Saulsbury, Fong Pong, and Andreas Nowatzyk, "Missing the Memory Wall: The Case for Processor/Memory Integration", In *Proceedings of the 23rd Annual International Symposium on Computer Architecture(ISCA),* pages 90-101, 1996.

[69] Ashley Saulsbury, Tim Wilkinson, John Carter, and Anders Landin, "An Argument for Simple COMA", In *Proceedings of the 1st International Symposium on High Performance Computer Architecture(HPCA),* January, 1995.

[70] Daniel J.Scales, Kourosh Gharachorloo, and Chandramohan A.Thekkath, " Shasta: a low overhead, software-only approach for supporting fine-grain shared memory", In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems(ASPLOS),* Pages 174 - 185, 1996.

[71] Ioannis Schoinas, Babak Falsafi, Alvin Lebeck, Steven Reinhardt, James Larus, and David Wood, "Fine-grain Access Control for Distributed Shared Memory", In *Proceedings of the 6th Conf. on Architecture Support for Programming Languages and Operating Systems(ASPLOS),* pages 297-306, 1994.

[72] Andre Seznec, "Don't Use the Page Number, But a Pointer To It," In *Proceedings of the 23rd Annual International Symposium on Computer Architecture(ISCA),* pages 104-113, 1996.

[73] Andre Seznec, "Decoupled Sectored Caches: Conciliating Low Tag Implementation Cost and Low Miss Ratio," In *Proceedings of the 21st Annual International Symposium on Computer Architecture(ISCA),* pages 265-274, 1994.

[74] Andre Seznec, " A Case for Two-Way Skewed-Associative Caches," In *Proceedings of the 20th Annual International Symposium on Computer Architecture(ISCA),* pages 169-178, 1993.

[75] James Smith and Andrew Pleszkun, "Implementation of Precise Interrupt in Pipelined Processors", In *Proceedings of the 12nd Annual International Symposium on Computer Architecture(ISCA),* pages 36-44, 1985.

[76] Yong Ho Song and Michel Dubois,"Assisted Execution", Technical Report #CENG 98-25, Department of EE-Systems, University of Southern California, October 1998.

[77] Thomas Sterling, Larry Bergman, "A Design Analysis of a Hybrid Technology Multithreaded Architecture for Petaflops Scale Computation", In *proceedings of the International Conference on Supercomputing(ICS'99),* Rhodes, Greece, June, 1999.

[78] Mark Swanson, Leigh Stoller, and John Carter, "Increasing TLB reach Using Superpages Backed by Shadow Memory", In *Proceedings of the 25th Annual International Symposium on Computer Architecture(ISCA),* pages 204-213, 1998.

[79] M. Talluri and M. D. Hill. "Surpassing the TLB Performance of Superpages with Less Operating System Support," In *Proceedings of the 6th Conf. on Architecture Support for Programming Languages and Operating Systems(ASPLOS),* 1994.

[80] M. Talluri, S. Kong, M. D. Hill, and D. A. Patterson. "Tradeoffs in Supporting Two Page Sizes," In *Proceedings of the 19th Annual International Symposium on Computer Architecture(ISCA),* pages 415-424, May 1992.

[81] Patricia Teller, "Translation Lookaside Buffer Consistency," Ph.D thesis.

[82] C. Thekkath and H. Levy, "Hardware and Software Support for Efficient Exception Handling," In *Proceedings of the 6th Conf. on Architecture Support for Programming Languages and Operating Systems(ASPLOS),* 1994.

[83] M. Tremblay and J. M. O'Connor, "Ultrasparc I: A Four-Issue Processor Supporting Multimedia," IEEE *Micro,* pages 42-50, April 1996

[84] D Tullsen, S. Eggers, and H. Levy, "Simultanous Multithreading: Maximizing On-Chip Parallelism," In *Proceedings of the 22nd Annual International Symposium on Computer Architecture(ISCA),* pages 392-403, 1995.

[85] W, H. Wang, J-L, Baer, and H. M. Levy, "Organization and performance of a two-level Virtual-Real cache hierarchy," In *Proceedings of the 16th Annual International Symposium on Computer Architecture(ISCA)*, pages 140-148, June 1989.

[86] Hong, Wang, Tong Sun, and Qing Yang, "CAT -- Caching Address Tags, A Technique for Reducing Area Cost of On-chip Caches", In *Proceedings of the 22nd Annual International Symposium on Computer Architecture(ISCA)*, page 381-390, 1995.

[87] David Weaver and Tom Germond, "The SPARC Architecture Manual", version 9, Prentice Hall, 1994.

[88] S. C. Woo, M. Ohara, and E. Torrie. "The SPLASH-2 Programs: Characterization and Methodological Considerations," In *Proceedings of the 22nd Annual International Symposium on Computer Architecture(ISCA)*, pages 24-36, 1995.

[89] David Wood, Susan Eggers, Garth Gibson, Mark Hill, and Joan Pendleton. "An In-Cache Address Translation Mechanism," In *Proceedings of the 13th Annual International Symposium on Computer Architecture(ISCA)*, pages 358-365, Jan. 1986.

[90] C. Eric Wu, Yarsun Hsu, and Yew-Huey Liu, " A Quantitative Evaluation of Cache Types for High-Performance Computer Systems", *IEEE Transactions on Computers*, pages 1154-1162, Vol 42, No. 10, Oct. 1993.

[91] W. A. Wulf, and Mckee, "Hitting the Memory Wall: Implications of the Obvious," *ACM Computer Architecture News*, Vol. 23, No. 1, March 1995.

[92] K. C. Yeager, "The MIPS R10000 Superscalar Microprocessor," IEEE *Micro*, pages 28-40, April 1996.

[93] Donald Yeung, John Kubiatowicz, and Anant Agarwal, "MGS: A Multigrain Shared Memory System", In *Proceedings of the 23rd Annual International Symposium on Computer Architecture(ISCA)*, pages 44-55, 1996.

[94] Craig Zilles, Joel Emer, and Gurindar Sohi, "The Use of Multithreading for Exception Handling,", In *Proceedings of the 32nd Annual International Symposium on Microarchitecture(Micro-32)*, 1999.