High-Bandwidth Address Translation for Multiple-Issue Processors

Todd M. Austin Gurindar S. Sohi

Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, WI 53706
{austin,sohi}@cs.wisc.edu

Abstract

In an effort to push the envelope of system performance, microprocessor designs are continually exploiting higher levels of instruction-level parallelism, resulting in increasing bandwidth demands on the address translation mechanism. Most current microprocessor designs meet this demand with a multi-ported TLB. While this design provides an excellent hit rate at each port, its access latency and area grow very quickly as the number of ports is increased. As bandwidth demands continue to increase, multi-ported designs will soon impact memory access latency.

We present four high-bandwidth address translation mechanisms with latency and area characteristics that scale better than a multi-ported TLB design. We extend traditional high-bandwidth memory design techniques to address translation, developing interleaved and multi-level TLB designs. In addition, we introduce two new designs crafted specifically for high-bandwidth address translation. Piggyback ports are introduced as a technique to exploit spatial locality in simultaneous translation requests, allowing accesses to the same virtual memory page to combine their requests at the TLB access port. Pretranslation is introduced as a technique for attaching translations to base register values, making it possible to reuse a single translation many times.

We perform extensive simulation-based studies to evaluate our designs. We vary key system parameters, such as processor model, page size, and number of architected registers, to see what effects these changes have on the relative merits of each approach. A number of designs show particular promise. Multi-level TLBs with as few as eight entries in the upper-level TLB nearly achieve the performance of a TLB with unlimited bandwidth. Piggyback ports combined with a lesser-ported TLB structure, *e.g.*, an interleaved or multi-ported TLB, also perform well. Pretranslation over a single-ported TLB performs almost as well as a same-sized multi-level TLB with the added benefit of decreased access latency for physically indexed caches.

1 Introduction

Address translation is a vital mechanism in modern computer systems. The process provides the operating system with the mapping and protection mechanisms necessary to manage multiple large and private address spaces in a single, limited size physical memory [HP90]. In practice, most microprocessors implement low-latency address translation with a translation lookaside buffer (TLB). A TLB is a cache, typically highly-associative, containing virtual memory page table entries which describe the physical address of a virtual memory page as well as its access permissions and reference status (*i.e.*, reference and dirty bits). The virtual page address of a memory access is used to index the TLB; if the virtual page address hits in the TLB, a translation is quickly returned. On a TLB miss, a hardware- or software-based miss handler is invoked which "walks" the virtual memory page tables to determine the correct translation to load into the TLB.

The primary goal of TLB design is to keep address translation latency off the critical path of memory access. In the past, this goal has been met by building low-latency TLBs. This task was relatively easy to perform because most TLB designs were single-ported and small, containing on the order of 32 entries.

Today, however, architectural and workload trends are placing increasing demands on TLB designs. Processor designs are continually exploiting higher levels of instruction-level parallelism (ILP), which increases the bandwidth demand on TLB designs. The nature of workloads is also changing. There is a strong shift towards codes with large data sets and less locality, resulting in poor TLB hit rates. Notable examples of this trend include environments that support multitasking, threaded programming, and multimedia applications.

Together, architectural and workload trends are pushing architects to look for TLB designs that possess low-latency and high-bandwidth access characteristics while being capable of mapping a large portion of the address space. The current approach used in most multiple-issue processors is a large multi-ported TLB, typically dual-ported with 64-128 entries. A multi-ported TLB provides multiple access paths to all cells of the TLB, allowing multiple translations in a single cycle. The relatively small size of current TLBs along with the layout of the highly-associative storage lends itself well to multi-porting at the cells [WE88].

Although a multi-ported TLB design provides an excellent hit rate at each access port, its latency and area increase sharply as the number of ports or entries is increased. While this design meets the latency and bandwidth requirements of many current designs, continued demands may soon render it impractical, forcing tomorrow's designs to find alternative translation mechanisms. Already, some processor designs have turned to alternative TLB organizations with better latency and bandwidth characteristics; for example, Hal's SPARC64 [Gwe95] and IBM's AS/400 64-bit PowerPC [BHIL94] processor both implement multi-level TLBs.

Our goal in this paper is to extend the work on high-bandwidth address translation in two ways. First, we propose a number of designs for high-bandwidth address translation with latency and area costs that scale better with the number of ports than a multi-ported TLB. Second, we perform extensive simulation-based studies to evaluate

Appears in: "Proceedings of the 23rd Annual International Symposium on Computer Architecture"

the relative merits of the proposed address translation designs.

Using detailed cycle-timing simulators, we benchmark the performance of the high-bandwidth designs against the performance of a TLB with unlimited bandwidth. A number of designs are clear winners – their use results in almost no impact on overall system performance. Any latency and area benefits these designs may afford will serve to improve system performance through increased clock speeds and/or better die space utilization.

We limit the scope of this work to translation of data memory accesses for physically tagged caches. Instruction fetch translation is a markedly easier problem, since fetch mechanisms typically restrict all instructions fetched in a single cycle to be within the same virtual memory page, requiring at most one translation per cycle. Instruction fetch translation is well served by a single-ported instruction TLB or by a small micro-TLB implemented over a unified instruction and data TLB [CBJ92].

The rest of this paper is organized as follows: Section 2 describes our framework for address translation and qualitatively explores the impact that address translation latency and bandwidth have on system performance. Section 3 details the mechanisms proposed for high-bandwidth address translation, and Section 4 presents extensive simulation-based performance studies of a number of address translation designs. Section 5 presents a summary and conclusions.

2 Impact of Address Translation on System Performance

Before delving into the details of our high-bandwidth designs or their evaluation, it is prudent to first develop a performance model for address translation. The model we present in this section is strictly qualitative in nature. We do not use it to derive the performance of a particular address translation mechanism; we do this empirically with detailed timing simulations in Section 4. Instead, the model serves as a framework for address translation. By casting our designs into this framework, we can readily see which features affect address translation performance, and consequently, how address translation performance affects system performance.

Figure 1 illustrates our framework for address translation. At the highest level, a processor core executes a program in which a fraction f_{MEM} of all instructions access memory. Each cycle, the processor core makes as many as M address translation requests. A fraction $f_{shielded}$ of these requests are serviced by a *shielding mechanism*. A shielding mechanism is a high-bandwidth and low-latency translation device that can satisfy a translation request without: 1) impacting the latency of memory access, or 2) forwarding the request to the base TLB mechanism. Hence, the shielding mechanism acts as a shield for the base TLB mechanism, filtering out $f_{shielded}$ of all translation requests. An effective shielding mechanism can significantly reduce the bandwidth demands on the base TLB mechanism; we examine three shielding mechanisms in detail: L1 TLBs, piggyback ports, and pretranslation.

Requests not handled by the shielding mechanism are directed to the base TLB mechanism which can service up to N requests per translation cycle. The base TLB mechanism functions identically to a traditional TLB, providing fast access to page table entries using a low-latency caching structure. However, the organization used in this paper may be non-traditional, *e.g.*, interleaved, for the purpose of providing increased bandwidth. If a base TLB port is immediately available, the translation proceeds immediately. If a port is not available, the request is queued until a port becomes available, at which time it may proceed. The queuing mechanism employed is dependent on the processor model, *e.g.*, an out-of-order issue processor queues requests in a memory re-order buffer, while an in-order issue processor queues requests by stalling the pipeline. Requests are queued waiting for a port for an average latency of $t_{stalled}$. The magnitude of $t_{stalled}$ is determined by the bandwidth of the address translation mechanism – with unlimited bandwidth $t_{stalled}$ will be

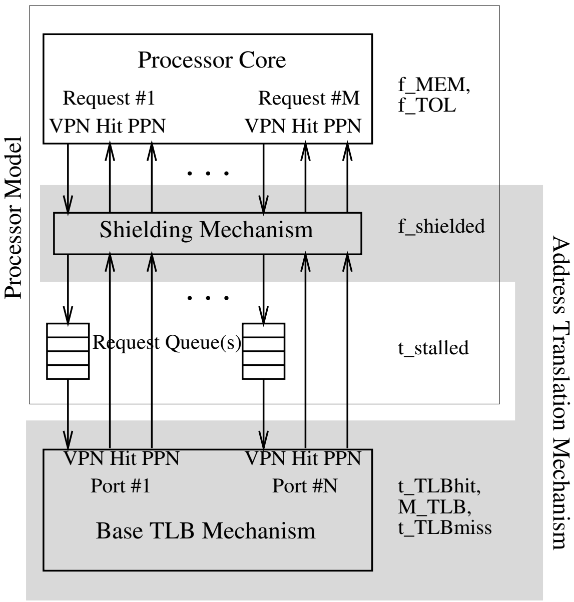

Figure 1: A System Model of Address Translation Performance. VPN is the virtual page number, PPN is the physical page number.

zero, with limited bandwidth it may be non-zero. How bandwidth affects queueing latency in the processor is very complex, since it depends on the frequency and distribution of requests to the translation device. We don't attempt to derive this relationship analytically. Instead, we measure precisely its impact through detailed timing simulations in Section 4. Once a request is serviced by the base TLB mechanism, $(1 - M_{TLB})$ requests will hit in the TLB and be serviced with latency t_{TLBhit}. The remaining M_{TLB} of all requests will miss in the TLB and be serviced with latency $t_{TLBmiss}$.

Under this model of address translation, the average latency of a translation request (as seen by the processor core), t_{AT}, is:

$$t_{AT} = (1 - f_{shielded}) * (t_{stalled} + t_{TLBhit} + M_{TLB} * t_{TLBmiss})$$

The effect of the latency of address translation seen by the processor core is tempered by two factors: 1) the processor's ability to tolerate latency, and 2) the relative impact of memory access latency compared to other latencies. Therefore, the system impact of address translation latency, measured as the average time per instruction due to address translation latency, TPI_{AT}, is:

$$TPI_{AT} = f_{MEM} * (1 - f_{TOL}) * t_{AT}$$

f_{TOL} is the fraction of address translation latency that is tolerated by the processor core. The workload and processor model both affect the degree to which the processor core can tolerate latency. If the workload exhibits sufficient parallelism and the execution model provides latency tolerating support, the impact of address translation latency on overall performance will decrease. Processor models with high levels of latency tolerating capability include those that support out-of-order issue, non-blocking memory access, and speculative execution.

Finally, f_{MEM} is the dynamic fraction of all instructions that access memory. This factor is affected by the workload, the number of architected registers, and the compiler's ability to effectively utilize registers. Programs that access memory often will need better address translation performance for good system performance.

In summary, the performance of the address translation mechanism is affected: 1) by its ability to shield requests from the base

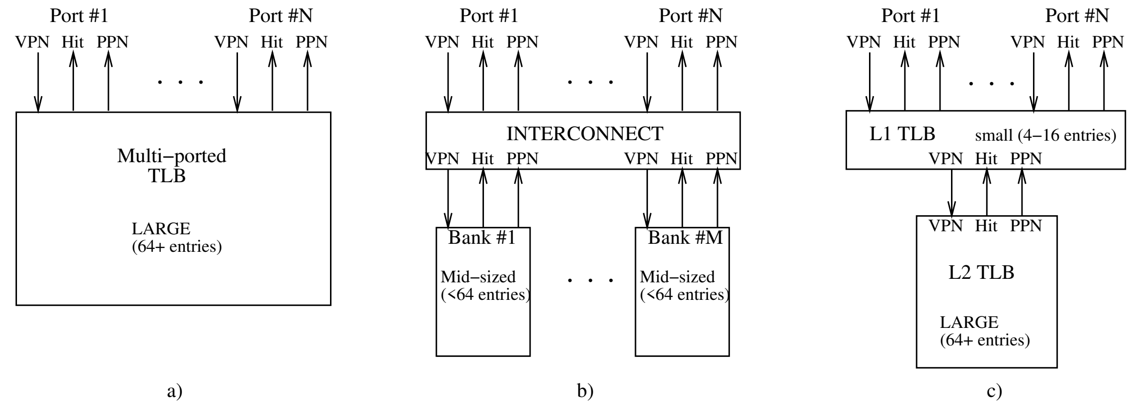

Figure 2: Traditional High-Bandwidth Memory Designs: a) multi-ported, b) interleaved, and c) multi-level.

translation mechanism, and 2) by the latency and bandwidth of the base translation device. The system impact of address translation performance is, however, affected by a program's reliance on memory access and the processor's ability to tolerate latency.

3 High-Bandwidth Address Translation

In this section, we present new mechanisms for high-bandwidth address translation. For each mechanism, we describe how it maps to our framework for address translation (shown in Figure 1) and highlight the strengths and weaknesses of the particular approach.

Our designs fall into two categories: 1) designs that extend traditional high-bandwidth memory design to the domain of address translation, and 2) designs crafted specifically for high-bandwidth address translation.

Techniques for delivering high-bandwidth memory access are well developed, both in the literature and in practice. The common approaches are multi-ported [SF91], interleaved [Rau91], and multi-level [JW94] memory structures. We can easily extend these approaches to the address translation domain.

Piggyback ports are introduced as a technique to exploit the high level of spatial locality in simultaneous translation requests. This approach allows simultaneous accesses to the same virtual memory page to combine their requests at the TLB access port. *Pretranslation* is introduced as a technique for attaching translations to base register values, making it possible to reuse a single translation many times.

All of our high-bandwidth address translation designs are targeted towards systems that use physically tagged caches, *i.e.*, those which require a translation for each memory access. Virtual address caches, however, do not require a translation for each memory access; address translation is pushed off until data is fetched from physical storage, *e.g.*, when a physically addressed second-level cache or main memory is accessed. Such a design eliminates both bandwidth and latency concerns. Virtual address caches have, however, two significant drawbacks which discourage their use in real systems: 1) synonyms, and 2) lack of support for protection.

Synonyms can occur in virtually indexed caches when storage is manipulated under multiple virtual addresses. In a multiprogrammed environment, shared physical storage can end up in multiple lines of a virtually indexed cache, creating a potential coherence problem. In a multiprocessing environment, cache coherence operations must first be reverse-translated to remote virtual addresses before remote data can be located in the remote cache. Many solutions have been devised to eliminate synonyms, including alignment restrictions on shared data [Che87], selective invalidation [WBL89], and single address space operating systems [KCE92]. However, these approaches have yet to come into widespread use due to performance and/or implementation impacts on application and system software. Moreover, these solutions do not solve the second problem that arises with virtual address caches, efficient implementation of protection.

Traditionally, protection information has been logically attached to virtual memory pages. As a result, their implementation has been naturally integrated into the TLB. If the TLB is eliminated through use of a virtual address cache, the problem of implementing protection still remains. One solution is to integrate protection information into cache blocks [Hea86]. However, the page-granularity of protection information makes managing these fields both complicated and expensive. Another solution is to implement a TLB minus the physical page address information [KCE92] – this TLB-like structure, however, still requires high-bandwidth and low-latency access (although, latency requirements are somewhat relaxed).

In light of these drawbacks, virtual address caches have seen little use in real systems. In addition, it is likely that if virtual address caches are adopted they may still employ TLB-like structures to implement protection, which requires a high-bandwidth mechanism like the ones we describe here. Consequently, we don't consider virtual address caches any further; instead, we concentrate on address translation designs for physically tagged caches.

3.1 Multi-ported TLB

A multi-ported TLB, illustrated in Figure 2a, uses a brute force approach to providing high-bandwidth. Each port is provided its own data path to every entry in the TLB, implemented by either replicating the entire TLB structure (one single-ported TLB for each port) or multi-porting the individual TLB cells. Since every entry of the TLB is accessible from each port of the device, this design provides a good hit rate for each port (low M_{TLB}). However, the capacitance and resistance load on each access path increases with the number of ports on the device [WE88], resulting in longer access latency (t_{TLBhit}) as the number of ports or entries increases. In addition, this design incurs a large area overhead due to the many extra wires and comparators needed to implement each port. (In CMOS technology, the area of a multi-ported device is proportional to the square of the number of ports [Jol91].)

Independent of access latency and implementation area considerations, this design provides the best bandwidth and hit rate of all the designs, hence, it provides a convenient standard for gauging the performance of the other approaches that we propose.

3.2 Interleaved TLB

An interleaved TLB, shown in Figure 2b, employs an interconnect to distribute the address stream among multiple TLB banks. Each TLB bank can independently service one request per transla-

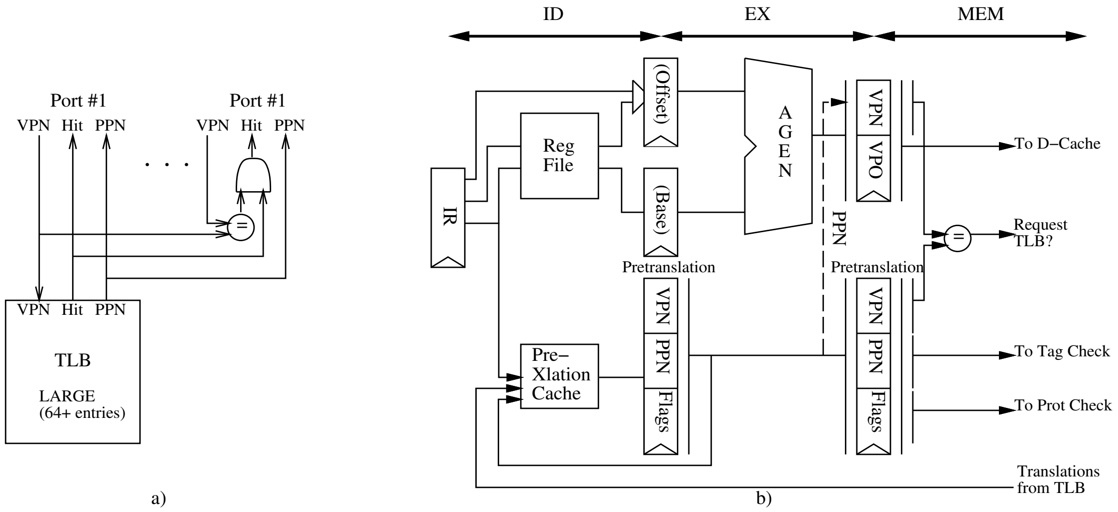

Figure 3: Address Translation Specific Designs: a) piggyback port, and b) pretranslation.

tion cycle. This design provides high-bandwidth access as long as simultaneous accesses map to different banks.

The mapping between virtual page addresses and the TLB banks is defined by the *bank selection function*. This function influences the distribution of the accesses to the banks, and hence, the bandwidth delivered by the device. In our evaluations, we consider both *bit selection*, which uses a portion of the virtual page address to select the bank, as well as an XOR-folding scheme, which randomizes the bank assignment by XOR'ing together portions of the virtual page address. (XOR-folding functions have been shown to provide better bank distribution [KJLH89].)

By its construction, an interleaved TLB may not be fully-associative, since any particular page may only reside in one bank. Its associativity must be limited to the associativity of the individual banks. As a result, M_{TLB} for this design may be higher than a same-size design with a more associative organization, possibly resulting in longer average translation latency. The impact should be minimal, however, if the interleaved TLB remains highly-associative.

This design will likely have better latency and area characteristics than a multi-ported TLB, especially for large TLBs. While the interconnect, typically a full crossbar, adds some latency to the access path, this latency is mitigated by the shorter access latency of the smaller, single-ported banks. The area overhead is concentrated in the interconnect; for a full crossbar, the implementation area is proportional to the square of number of access ports. For small numbers of ports, sizes should not be prohibitively large.

3.3 Multi-level TLB

A multi-level TLB, shown in Figure 2c, provides high-bandwidth and low-latency address translation by exploiting locality in program references. When an entry from the base TLB mechanism (L2 TLB) is referenced, it is placed into in a small upper-level TLB (L1 TLB). An L1 TLB acts as a shielding mechanism; if it offers a good hit rate, it will shield the L2 TLB from all accesses that hit in the L1 TLB, significantly reducing the bandwidth demand on the L2 TLB.

When an access misses in the L1 TLB, it must forward the request to the L2 TLB, where L2 TLB access port contention, L2 TLB access latency, and L2 TLB miss latency may increase overall the latency of the access. Since the L1 TLB is small, it may be possible to use a more effective replacement policy (*e.g.*, LRU replacement in the L1 TLB vs. random replacement in the L2 TLB), which should improve the hit rate of the L1 TLB.

If the processor supports hardware-based TLB consistency operations [BRG$^+$89], multi-level inclusion should be enforced in the L1 TLB during L2 TLB replacements or invalidations, *i.e.*, the entries in the L1 TLB should be a subset of the entries in the L2 TLB. This implementation strategy will eliminate the need for consistency operations to probe the L1 TLB, which may be expensive if it is tightly integrated into the processor pipeline.

The L1 TLB is a multi-ported TLB with enough ports to handle all simultaneous requests from the processor core. By keeping the L1 TLB small, it is possible to provide both high-bandwidth and low-latency access to all its entries. The additional area overhead of this design is concentrated in the implementation of the L1 TLB, which for small sizes and few ports should be much smaller than the L2 TLB.

At least two commercial processors have explored the use of multi-level TLBs; Hal's SPARC64 [Gwe95] and IBM's AS/400 64-bit PowerPC [BHIL94] processors both implement multi-level TLBs to meet the latency and bandwidth needs of their respective designs. Multi-level TLB designs have long been used for reducing the latency of instruction fetch translations [CBJ92].

3.4 Piggyback Ports

Piggyback ports, shown in Figure 3a, exploit spatial locality in simultaneous address translation requests. When simultaneous requests arrive at a TLB port, requests with identical virtual page addresses may be satisfied by the same TLB access.

To implement piggybacking, the virtual page addresses of blocked requests are compared to the virtual page address of requests in progress. A blocked request may use the result of a translation in progress if their virtual page addresses match. For a single port, the hit detection signal from the TLB port can be gated with the result of the virtual page address comparison. The approach is similar to read combining in multiprocessor interconnection networks [LS94]. Assuming both requests are executing under the same protection domain, the other fields of the translation request, *i.e.*, protection and page status information, may also be forwarded to other requesters with matching virtual page addresses.

Piggyback ports have minimal impact on translation latency. Once a request is submitted to the TLB, all other requesters can compare virtual addresses in parallel with TLB access. As a result, the impact on translation latency is limited to the gating of the TLB hit signal. Area costs are also very small, being limited to a single comparator and hit signal gate per piggyback port.

3.5 Pretranslation

Pretranslation is a shielding mechanism that allows a single translation request to be used for multiple memory accesses. Figure 4 illustrates the basis for this approach. Loads and stores access memory through register pointers: global accesses through the global pointer [CCH+87], stack accesses through the stack pointer, and all other references through general purpose register pointers. Pointers are created whenever a variable is referenced, its address is taken, or when dynamic storage is allocated. During the lifetime of a pointer, it is dereferenced at loads and stores, and manipulated using integer arithmetic. Over the lifetime of the pointer, it may be dereferenced and manipulated many times.

Studies have shown, *e.g.* [EV93], that when pointers are manipulated, it is often the case that small constant values are added to or subtracted from the pointer. The end result, which we exploit in this design, is that translations between successive uses of a pointer often yield accesses to the same virtual memory page.

In traditional TLB-based address translation mechanisms, an address translation request is made to the TLB each time a pointer is dereferenced, often requesting the same translation on subsequent requests. With pretranslation, we attach a translation to a register *value* at the first dereference of the value, *i.e.*, at the first load or store to use the register as a base register value. On subsequent dereferences, loads and stores may use the translation (or as we term it, pretranslation) attached to the register value provided that the virtual page address of the memory access matches the virtual page address of the attached translation. When pointers are manipulated with arithmetic operations, any attached translation is propagated to the destination register value. Pretranslation yields high bandwidth as long as register pointers are reused often and point to the same virtual memory page. Thus, a single translation request from the base TLB mechanism may be used multiple times.

Our pretranslation design is shown integrated into a processor pipeline in Figure 3b. Pretranslations are accessed in parallel with register file access in the decode stage of the pipeline, making the pretranslation available by the start of instruction execution. If the instruction is an arithmetic operation, the pretranslation is attached to the result register value. For loads and stores, the pretranslation, if available, is used to elide TLB access if the virtual page addresses match. If the virtual page addresses do not match, a translation request is forwarded to the base translation mechanism. The result of the translation is attached to the base register value.

Two important considerations affect the design of the mechanism used to attach pretranslations to register values. First, a single pointer value may reference multiple pages. A suitable mechanism to attach multiple translations to a single register may improve performance, *e.g.*, a few bits from the offset could be combined with the base register identifier to form the identifier of a pretranslation. Second, only a fraction of all registers will be pointer values at any one time, thus, storage need not be allocated for each register. It suffices to use a small cache (which we term a pretranslation cache) to hold pretranslations. If this cache is kept small, it will facilitate high-bandwidth and low-latency access to pretranslations.

Any changes in virtual memory state, *e.g.*, address mapping, page size, or access permission, must be reflected in the pretranslation cache, otherwise, invalid accesses may go undetected. If virtual memory state changes are infrequent, it may be sufficient to simply flush the pretranslation cache whenever changes occur.

The VAX IPA register used a similar technique to reuse a translation for instruction fetching [LE89]. The current PC physical address translation is stored in the Instruction Physical Address (IPA) register, and this translation is used to access the cache until: 1) the PC crosses a page boundary, or 2) a branch is taken. On either of these events, the previous translation is invalidated and another address translation of the PC is initiated. Bray's translation hit buffer (THB) [BF92] further extends this idea to include a prediction of the

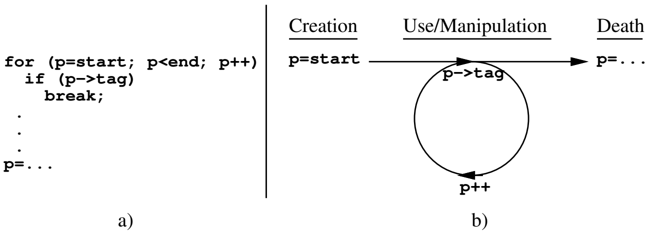

Figure 4: The Life of a (Register) Pointer. Figure a) shows a C code fragment in which pointer `p` strides through an array. Figure b) illustrates the operations that occur over the lifetime of pointer `p`.

next translation as well.

Pretranslation can be viewed as an extension of Chiueh and Katz's *branch address cache* (BAC) [CK92], which was applied as a mechanism to reduce access latency of physically indexed caches. (A similar mechanism was proposed in [HHL+90].) Our design extends the BAC technique to provide high-bandwidth translation. By attaching the virtual page address to a register value, the base TLB mechanism does not have to be accessed to validate use of an attached physical page address. Like the BAC, our design provides the physical page address by the end of instruction decode. Thus, it may be used to access a physically indexed cache without an added latency for address translation.

Our design includes two modifications to the original BAC mechanism. First, our design tracks instructions that create pointer values, and propagates the pretranslation of any operand to the result register. This optimization is important for good performance on optimized code where register copies occur often, for example, during instruction scheduling or loop unrolling. Second, we employ a small cache to store pretranslations, instead of the larger BAC. Since only a fraction of all registers contain pointer values at any one time, our small pretranslation cache provides an excellent hit rate.

4 Experimental Evaluation

We evaluated the relative merits of our high-bandwidth address translation designs by extending a detailed timing simulator to support the proposed translation mechanisms and by examining the performance of programs running on the extended simulator. We varied the page size, processor issue model, and number of architected registers to see what affect these system parameters had on the translation mechanisms. All the results presented in this section are run-time weighted averages across all the benchmarks. Individual results for all experiments are available via FTP in the file "`ftp://ftp.cs.wisc.edu/sohi/isca96-results.ps.Z`".

4.1 Methodology

All programs were compiled with GNU GCC (version 2.6.2), GNU GAS (version 2.5), and GNU GLD (version 2.5) with maximum optimization (-O3) and loop unrolling enabled (-funroll-loops). The Fortran codes were first converted to C using AT&T F2C version 1994.09.27. All experiments were performed on an extended (virtual) MIPS-like architecture. The architecture implements a superset of the MIPS-I instruction set [KH92], with the following modifications:

- extended addressing modes: `register+register` and post-increment and decrement are included
- no architected delay slots

Our baseline simulator is detailed in Table 1. The simulator executes only user-level instructions, performing a detailed timing simulation of an 8-way superscalar microprocessor and the first level of instruction and data cache memory. The simulator supports both

Fetch Interface	fetches any 8 instructions in same cache block per cycle, separated by at most one branch
Instruction Cache	32k 2-way set-associative, 32 byte blocks, 6 cycle miss latency
Branch Predictor	8 bit global history indexing a 4096 entry pattern history table (GAp [YP93]) with 2-bit saturating counters, 3 cycle misprediction penalty
In-Order Issue Mechanism	in-order issue of up to 8 operations per cycle, allows out-of-order completion
Out-of-Order Issue Mechanism	out-of-order issue of up to 8 operations per cycle, 64 entry re-order buffer, 32 entry load/store queue, loads may execute when all prior store addresses are known
Architected Registers	32 integer, 32 floating point
Functional Units	8-integer ALU, 4-load/store units, 4-FP adders, 1-integer MULT/DIV, 1-FP MULT/DIV
Functional Unit Latency (total/issue)	integer ALU-1/1, load/store-2/1, integer MULT-3/1, integer DIV-12/12, FP adder-2/1, FP MULT-4/1, FP DIV-12/12
Data Cache	32k 2-way set-associative, write-back, write-allocate, 32 byte blocks, 6 cycle miss latency, four-ported, non-blocking interface, supporting one outstanding miss per physical register
Virtual Memory	4K byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete

Table 1: Baseline Simulation Model.

Mnemonic	Description
T4	4-ported TLB, 128 entries, fully-associative, random replacement
T2	2-ported TLB, 128 entries, fully-associative, random replacement
T1	1-ported TLB, 128 entries, fully-associative, random replacement
I8	8-way bit-select interleaved TLB, 128 entries (16 entry fully-associative bank), random replacement in bank
I4	4-way bit-select interleaved TLB, 128 entries (32 entry fully-associative bank), random replacement in bank
X4	4-way XOR-select interleaved TLB, 128 entries (32 entry fully-associative bank), random replacement in bank
M16	4-ported 16-entry L1 TLB w/LRU replacement, 128-entry L2 TLB, fully-associative, random replacement
M8	4-ported 8-entry L1 TLB w/LRU replacement, 128-entry L2 TLB, fully-associative, random replacement
M4	4-ported 4-entry L1 TLB w/LRU replacement, 128-entry L2 TLB, fully-associative, random replacement
P8	4-ported 8-entry pretranslation cache w/LRU replacement, 128-entry L2 TLB, fully-associative, random replacement
PB2	2-ported TLB w/ 2 piggyback ports, 128 entries, fully-associative, random replacement
PB1	1-ported TLB w/ 3 piggyback ports, 128 entries, fully-associative, random replacement
I4/PB	4-way bit-select interleaved TLB w/piggybacked banks, 128 entries (32 entries/bank), random replacement in bank

Table 2: Analyzed Address Translation Designs.

in-order and out-of-order issue execution models. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or misprediction. The in-order issue model provides no renaming and stalls whenever any data hazard occurs on registers. The out-of-order issue model employs a 64 entry re-order buffer that implements renamed register storage and holds results of pending instructions. Loads and stores are placed into a 32 entry load/store queue. Stores execute when all operands are ready; their values, if speculative, are placed into the load/store queue. Loads may execute when all prior store addresses have been computed; their values come from a matching earlier store in the store queue or from the data cache. Speculative loads may initiate cache misses if the address hits in the TLB. If the load is subsequently squashed, the cache miss will still complete. However, speculative TLB misses are not permitted. That is, if a speculative cache access misses in the TLB, instruction dispatch is stalled until the instruction that detected the TLB miss is squashed or committed. Each cycle the re-order buffer commits up to 8 results in-order to the architected register file. When stores are committed, the store value is written into the data cache. The data cache modeled is a four-ported 32k two-way set-associative non-blocking cache.

We found early on that instruction fetch bandwidth was a critical performance bottleneck. To mitigate this problem, we implemented a limited variant of the collapsing buffer described in [CMMP95]. Our implementation supports two predictions per cycle within the same instruction cache block, which provides significantly more instruction fetch bandwidth and better pipeline resource utilization.

A number of changes were made to the simulator to support our high-bandwidth address translation mechanisms. The designs we examine, with their mnemonic designations, are listed in Table 2.

For all configurations, TLB access is assumed to be fully overlapped with data cache access. Thus, address translation does not create a visible latency unless the translation mechanism cannot immediately service a translation request, *i.e.*, due to insufficient TLB bandwidth or a TLB miss. When multiple requests meet at a single TLB port, the port is allocated first to the earliest issued instruction. The interleaved schemes, *i.e.*, I8 and I4, use bit selection to select the TLB bank; the three or two address bits immediately above the page offset portion of the virtual address are used to select the proper TLB bank. The configuration X4 uses an XOR-folding of the three least significant groups of two address bits immediately above the page offset portion of the virtual address. In the two-level designs, *i.e.*, M16, M8, and M4, the L1 TLB can service up to four hits per cycle. L1 TLB misses are sent in the following cycle to the L2 TLB, where they may queue if other requests are being serviced by the L2 TLB. The minimum latency for an L1 TLB miss is 2 cycles. The pretranslation cache design (P8) has a hit latency of one cycle; misses are not detected until the cycle immediately following address generation, resulting in at least one more cycle latency for access to the single-ported base TLB. Like the multi-level TLB designs, requests to the single-ported base TLB may have to queue waiting for the port. The pretranslation cache tags are composed of the register identifier (5 bits) concatenated with the upper 4 bits of the offset of a load or zero for any other instruction. In the piggybacked designs, *i.e.*, PB2 and PB1, requests that do not receive a translation port may piggyback off any other translation performed in the same cycle. For the I4/PB configuration, piggyback ports are provided at each bank of the TLB, thus, simultaneous requests that meet at the same bank may be ser-

Program	Inputs/Options	Insts (Mil.)	Loads (Mil.)	Stores (Mil.)	Inst/Cycle Issue	Inst/Cycle C'mit	(Ld+St)/Cycle Issue	(Ld+St)/Cycle C'mit	Br Pred Rate (%)
Compress	in	62.0	15.8	6.1	3.65	1.96	1.30	0.69	89.7
Doduc	doducin	1,375.1	330.4	130.2	2.16	1.76	0.71	0.59	86.6
Espresso	-t cps.in	517.5	116.5	32.7	4.48	2.90	1.32	0.84	90.2
GCC	-O 1stmt.i	110.6	26.4	16.5	3.56	1.87	1.32	0.72	80.2
Ghostscript	-dNOPAUSE -sDEVICE=ppm fast-addr.ps -c quit	625.2	109.1	53.3	2.76	2.18	0.73	0.55	93.3
MPEG_play	coil.mpg	529.6	114.9	47.9	4.10	2.82	1.19	0.87	85.9
Perl	tests.pl	231.5	57.7	37.2	2.85	1.43	1.10	0.57	81.2
TFFT	MEXPONENT=20, ITER=1	959.8	136.6	89.4	2.69	1.79	0.62	0.42	79.9
Tomcatv	N=129	359.7	90.9	18.3	3.64	2.72	1.00	0.83	86.6
Xlisp	li-input.lsp	962.7	289.2	171.6	4.17	2.52	1.86	1.21	87.9

Table 3: Program Execution Performance. Instruction, load, and store counts include only non-speculative operations. The columns labeled *Issue* and *C'mit* indicate the average number of operations issued and committed per cycle, respectively, on the baseline 8-way out-of-order issue processor simulator.

viced at the same time if their virtual page addresses match.

In the multi-level TLBs and pretranslation design, *i.e.*, M16, M8, M4, and P8, page status information (*i.e.*, reference and dirty bits) is propagated into the upper-level caching structures. However, when a change must be made to the page status (*e.g.*, first reference or write to a page), the change is immediately sent to the base TLB, where the access may be queued if a port is not available immediately. This write-through strategy for page status information simplifies flushing of the upper-level TLB structure, since any status in the upper-level cache structure is fully replicated in the base TLB. Immediately propagating page status changes to the base TLB has little impact on performance, because page status changes require little bandwidth. Multi-level inclusion is enforced in the L1 TLBs, *i.e.*, M16, M8, and M4, by loading TLB misses into both the L1 TLB and the L2 TLB, and by selectively invalidating from the L1 TLB any entry replaced in the L2 TLB. Coherence is enforced in the pretranslation cache by flushing it whenever an entry in the base TLB is replaced.

4.2 Analyzed Programs

When selecting benchmarks, we looked for programs with varying memory system performance, *i.e.*, programs with large and small data sets as well as high and low reference locality. Table 3 details the programs we analyzed (giving their inputs, and instruction and reference counts) and the corresponding performance on the baseline simulator. *Compress*, *Doduc*, *Espresso*, *Tomcatv*, and *Xlisp* are from the SPEC '92 benchmark suite. *Ghostscript* is a postscript viewer rendering a page with text and graphics to a PPM-format graphics file. *MPEG_play* is an MPEG video decoder displaying a 79 frame compressed video file. *Perl* is a script language interpreter running its test suite. *TFFT* performs real and complex FFTs on a randomly generated data set. *Ghostscript* and *TFFT* have the largest data sets, roughly 10 and 40 Mbytes, respectively. *Compress*, *MPEG_play*, and *TFFT* have notably little locality in their reference streams; small data caches and TLBs perform very poorly for these three programs.

4.3 Baseline Performance

Figure 5 shows the performance of all the designs running on the baseline processor model, an aggressive 8-way out-of-order issue processor with 32 registers and 4k virtual memory pages. The run-time weighted average IPC (weighted by the run-time of T4 in cycles) is shown for each design. The IPCs are normalized to the IPC of the four-ported TLB design (T4). The T4 design provides a convenient benchmark, since it can service up to four translation requests per cycle, thus no latency is introduced into the results due to insufficient translation bandwidth. (The baseline simulator has a four-ported data cache, so cache bandwidth is never a bottleneck.) Since the timing simulations only count cycles, any clock cycle effects that a poorly scalable design (such as T4) might introduce are

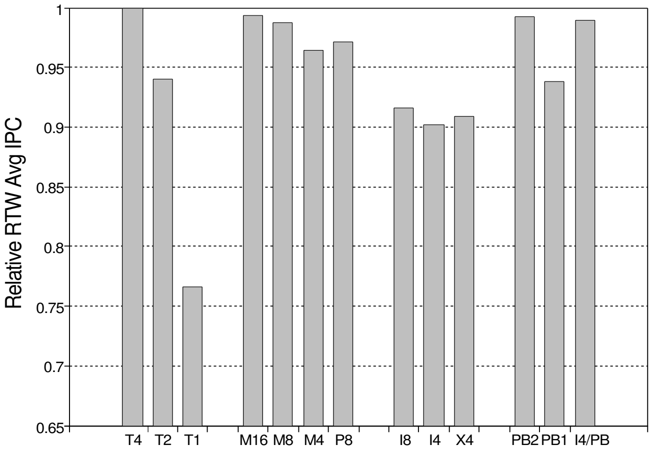

Figure 5: Relative Performance on Baseline Simulator. All results are run-time weighted average IPCs normalized to the performance of design T4.

ignored. On this common ground, the relative performance of a particular design indicates the cycle time improvement required to make the design worth implementing. For example, the average IPC of the 2-ported TLB design (T2) is 94.1% of the 4-ported design (T4), as a result, for a T2-based design to be a win, the average time per instruction must be at least 0.941 times that of the T4 design.

The leftmost group of bars in Figure 5 are the multi-ported TLB designs, *i.e.*, T4, T2, and T1, with 4, 2, and 1 port(s), respectively. These results demonstrate how sensitive the simulated system is to address translation bandwidth. Since the four-ported TLB design (T4) provides all the translation bandwidth the processor needs, its performance is always the best. With half as much translation bandwidth, *i.e.*, the dual-ported TLB (T2), the average IPC drops by 6%. With a single-ported TLB (T1), performance drops off sharply to 76% of the performance of the four-ported TLB (T4) design. Clearly, to not impact system performance, a translation device will have to provide at least two translations per cycle.

The second group of bars in Figure 5 are the multi-level (*e.g.*, M16, M8, and M4) and pretranslation (P8) designs. The performance of multi-level TLBs is quite good. An L1 TLB with as few as four entries over a single-ported L2 TLB suffers less than a 4% degradation in average IPC. Figure 6 indicates why the multi-level designs perform so well. This figure shows the run-time weighted average miss rates (labeled RTW Avg) for fully-associative TLBs from 4 to 128 entries. The 4, 8, and 16 entry TLBs use LRU replacement (as done for the 4, 8, and 16 entry L1 TLBs), while the 32, 64, and 128 entry TLBs employ random replacement (as done for the

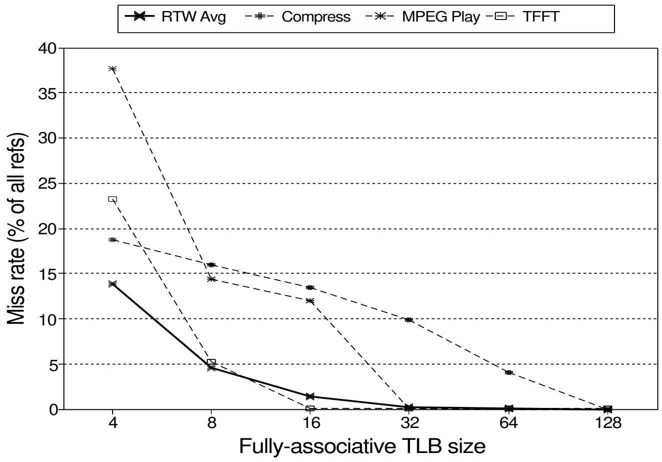

Figure 6: TLB Miss Rates. All values shown indicate percent of all references that miss in a fully-associative TLB. The line labeled RTW Avg is the run-time weighted average miss rate over all the benchmarks.

128 entry base TLB mechanisms). A four entry L1 TLB with LRU replacement shields all but 13.8% of the translation requests from reaching the L2 TLB. This shielding effect significantly reduces the bandwidth demand on the L2 TLB. The few references that do reach the L2 TLB have only slightly longer latency which is effectively tolerated by the out-of-order issue processor. A few of the programs, most notably *Compress*, *MPEG_play*, and *TFFT*, have poor performance on the multi-level designs. These programs have very low locality in the data reference stream, as can be seen by their large TLB miss rates in Figure 6.

While the pretranslation design (P8) performs well, *i.e.*, less than a 3% degradation in average IPC, its overall performance is worse than a same-sized L1 TLB. The reason for this difference lies in the mechanism by which each design reuses translations. The pretranslation design is only able to reuse a translation whenever a register pointer is reused. The multi-level TLB design, on the other hand, is able to reuse a translation in the L1 TLB whenever an address is reused. The latter case is more common, since when a new register pointer is first used on the pretranslation designs it must be translated, while on the multi-level designs, the address the new register pointer creates may be in the L1 TLB. It is interesting to note that reference locality and register reuse are sometimes orthogonal. In a few specific instances, *e.g.*, *Compress* and *GCC*, the pretranslation designs performed better than a same-sized L1 TLB. This contradictory behavior is likely due to better cache management for the pretranslation design. When new pointer values are created, they are reinserted into the pretranslation cache, which places the entry on the tail of the LRU queue. Other benefits of the pretranslation cache, such as early presentation of the physical page address should further motivate the use of this design. (Our simulations do not take advantage of early presentation of the physical page address.)

The interleaved designs did not perform as well as the multi-level designs, providing on the average less bandwidth than a dual-ported TLB (T2). This rather lackluster performance was not due to the set-associative organization required by the interleaved configurations. All of the configurations analyzed were at least 16-way set-associative and possessed excellent hit rates. Poor performance was due to bank conflicts which delayed requests. Increasing the number of banks (I8) or use of an XOR-folding bank selection function (X4) provided only marginal benefit, suggesting that many simultaneous accesses were to the same page, thus no increase in interleaving or change in bank selection function could eliminate conflicts.

The piggybacked designs, *i.e.*, PB2 and PB1, performed better than the interleaved designs. Piggybacking a single-ported TLB (PB1) resulted in only a 6% worse average IPC than the four-ported TLB design (T4). Clearly, many simultaneous accesses are to the same virtual page. However, not all concurrent accesses reference the same page as seen in the improved performance of the piggybacked dual-ported TLB design (PB2). This design can perform two independent translations per cycle, all other requests may use the result of either translation. The piggybacked dual-ported TLB design (PB2) performs nearly as well as the four-ported TLB design (T4).

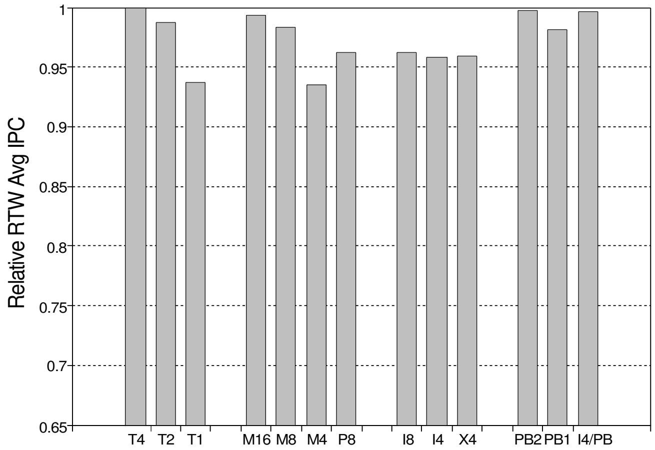

Figure 7: Relative Performance with In-order Issue.

Design I4/PB is an interleaved TLB with piggyback ports at each bank. This design leverages off the complementary benefits of the interleaved and piggybacked approaches. For an address stream with little spatial locality, requests will be steered to different banks and be serviced in parallel. For an address stream with good spatial locality, requests to the same page will be steered to the same bank and can share the translation result using the piggyback ports. This design should account for only a minimal increase in translation latency, since the addition of the piggyback ports only adds a single gate to the hit detection signal. (The virtual page address comparison to determine if the translation may be piggybacked occurs in parallel with bank access.) As shown in the Figure 5, this design performs very well, resulting in only a 1% degradation in average IPC.

4.4 Impact of In-Order Issue Model

Figure 7 shows the performance of the designs under the same conditions as Figure 5 except the processor is constrained to use an in-order issue model. This modification has two competing effects on the results. First, the average IPC of the in-order issue processor is markedly lower than that of the out-of-order issue processor, *i.e.*, 1.156 vs. 2.094, respectively. Consequently, the bandwidth demand on the address translation mechanism is reduced. Second, the in-order issue processor model cannot tolerate latency as effectively as the out-of-order issue processor. Thus, it is much more sensitive to address translation latency introduced by insufficient bandwidth.

Figure 7 shows the results of the experiments running on the 8-way in-order issue processor. The multi-ported TLB designs, *i.e.*, T4, T2, and T1, demonstrate the reduced bandwidth demand on the address translation. With only a single-ported TLB (T1), performance only degrades 6% below performance with a four-ported TLB (T4). The multi-level designs still perform well, although the performance of the 4 entry L1 TLB (M4) was affected more by the in-order issue model than the 8 (M8) and 16 (M16) entry designs. This result is likely due to the reduced latency tolerating capabilities of the in-order issue model, which cannot tolerate the 2 or more cycle latency incurred for the 13.8% of all memory accesses that must be serviced by the L2 TLB. The out-of-order issue model tolerates this latency much better than the in-order model, resulting in better overall performance. The interleaved designs perform much better on the

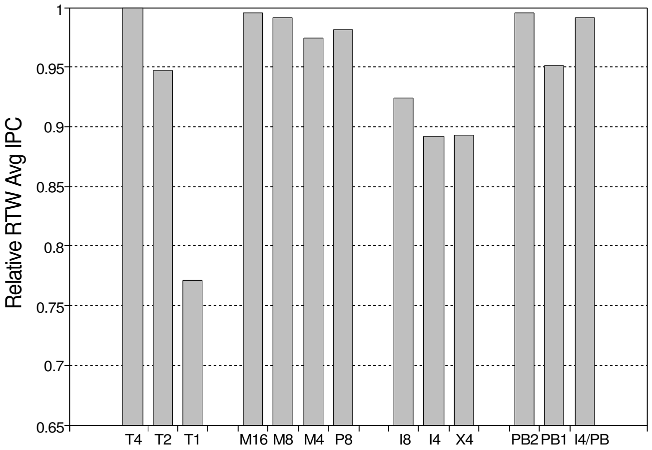

Figure 8: Relative Performance with 8k Pages.

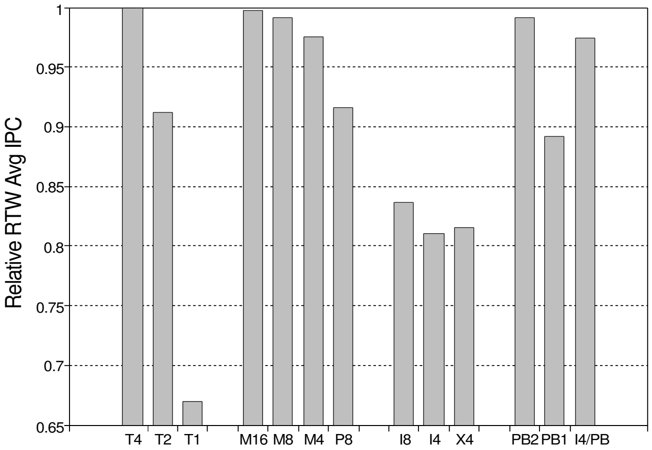

Figure 9: Relative Performance with Fewer Registers (8 int/8 fp).

in-order issue model. The degradation in IPC dropped from 10% to about 5% for these experiments. The reduced bandwidth demands on the interleaved designs reduces the number of bank conflicts. The piggybacked designs all perform better, with the PB2 and I4/PB designs experiencing virtually no degradation in average IPC.

4.5 Impact of Increased Page Size

A recent trend in TLB design has been to increase page sizes [TH94]. This trend is prompted by workloads with large data sets and/or little locality. Increased page size has a number of effects on the performance of the designs. With the same number of TLB entries, more memory may be mapped, which can reduce the number of TLB misses for both the base and L1 TLBs. Increased page size will increase the lifetime of pretranslations, allowing a pointer to stride further before leaving a page. Larger pages will also affect bank selection in the interleaved TLB designs, address bits formerly used to select the bank will become part of the page offset of the larger page. Changing the bank selection function will affect the distribution of accesses to the TLB banks.

Figure 8 shows the performance of the translation mechanisms running on the baseline 8-way out-of-order issue processor, except with 8k pages instead of 4k pages. The performance of the multi-ported designs is mostly unchanged, because the TLB miss rates were unchanged. The miss rates with a 128 entry TLB with 4k pages are already very low. The multi-level and pretranslation designs benefited from the larger page size. The L1 TLBs can map more memory and hence have better hit ratios, while the pretranslation cache benefited from longer pretranslation lifetimes. The interleaved designs performed roughly the same as with 4k pages, although there were some large variations in individual program performance due to changes in the bit selection function. As expected, the larger page size improved the performance of the piggybacked designs, *i.e.*, PB2 and PB1 and I4/PB, since the larger page size provides more opportunity to piggyback requests.

4.6 Impact of Fewer Registers

A number of architectures in wide-spread use today have few architected registers, *e.g.*, the *x86* or *System/370* architectures. To evaluate the efficacy of our high-bandwidth translation mechanisms for these architectures, we measured the performance of the benchmarks recompiled to use only 8 integer and 8 floating point registers (one-quarter the normal supply). The primary effect of fewer registers is an increased number of loads and stores, as many as 346% more for *Tomcatv*. Most of these references are directed to the stack and global regions of the data memory address space with a high degree of spatial and temporal locality. The results of the experiments are shown in Figure 9. All simulations were performed on the baseline 8-way out-of-order issue processor with 4k pages.

Even with the many extra memory accesses, the multi-level designs perform well. However, the pretranslation design (P8) suffered because (with few registers) pointer register value lifetimes were severely shortened due to many extra spills. When a pointer is spilled to the stack its pretranslation is lost, thus, another translation request must be made to the single-ported base TLB when it is reloaded. The performance of the interleaved designs was impacted significantly, dropping nearly 10% overall. Comparing the multi-level performance to the interleaved designs supports the conclusion that the many extra references have a high degree of locality. However, as shown by the poor performance of the piggybacked single-ported TLB designs (PB1), the locality is not always to the same virtual memory page. The interleaved and piggybacked design (I4/PB) performs slightly worse, suggesting that the extra accesses may have spatial locality spanning a page, which could occur for very large stack frames or many extra accesses directed to a large global region.

5 Summary and Conclusions

Four alternative mechanisms for high-bandwidth address translation were presented: interleaved TLBs, multi-level TLBs, piggyback ports, and pretranslation. These translation mechanisms all have latency and area characteristics that scale better than a simple multi-ported TLB, providing architects with better design choices as architectural and workload trends make it increasingly difficult to rely on a multi-ported TLB for good performance.

We performed extensive evaluations of a number of designs employing these basic high-bandwidth mechanisms. We examined their performance in a number of contexts: with out-of-order and in-order issue processors, with large and small pages, and on architectures with many and few registers.

Overall, we found several designs performed on par with a four-ported TLB. The multi-level TLB designs performed well except for programs with poor reference locality. The interleaved and piggybacked designs complement each other; an interleaved TLB with piggybacking at each bank performed well for all programs. Alone, the interleaved designs performed poorly due to many simultaneous accesses to the same bank, which without support for piggybacking are serialized at the bank. Piggybacking alone also performed poorly over a single-ported TLB due to many accesses occurring simultaneously to different pages. A piggybacked dual-ported TLB appears to be an adequate substitute for a four-ported TLB.

The pretranslation design also performed well, although its performance was slightly worse than a same-sized multi-level TLB design. Other benefits of this design should motivate its use. Pretranslations are available early in the pipeline, facilitating the use of upper-level physically indexed caches. In addition, attaching address information to physical registers prior to reception of their results could have other benefits, *e.g.*, classifying computation as access and execute, or using the address information to disambiguate memory references.

With in-order issue, bandwidth demand on the translation mechanism is reduced, but it still must perform well to provide good system performance due to the reduced latency tolerating capability of the in-order issue processor. The reduced bandwidth appears to be the stronger force, resulting in better overall performance for all the translation designs.

With larger pages (*i.e.*, 8k vs. 4k), the multi-level, pretranslation, and piggybacked designs performed well. Larger pages allow the L1 TLBs to map more address space and benefit pretranslation because pointers may stride further before a pretranslation is invalidated.

With few registers (*i.e.*, 8 int/8 fp vs. 32 int/32 fp), bandwidth demands on the translation mechanism rose sharply. All but the multi-level designs suffered worse performance. The high degree of reference locality in the extra references generated allowed a small L1 TLB to service most of the load. Pretranslation performed worse with fewer registers due to shorter register lifetimes.

Clearly, there exist many effective alternatives to the brute force approach of multi-porting the TLB. The designs presented in this paper should give architects plenty of choices when multi-ported TLB designs become impractical.

Acknowledgements

We thank Scott Breach, Dionisios Pnevmatikatos, and the referees for their comments on drafts of this paper. This work was supported in part by NSF Grants CCR-9303030 and MIP-9505853, ONR Grant N00014-93-1-0465, a donation from Intel Corp., and by U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U. S. Army Intelligence Center and Fort Huachuca, or the U.S. Government.

References

[BF92] B. K. Bray and M. J. Flynn. Translation hint buffers to reduce access time of physically-addressed instruction caches. *Proc. of the 25th Annual International Symposium on Microarchitecture*, 23(1):206–209, December 1992.

[BHIL94] J. Borkenhagen, G. Handlogten, J. Irish, and S. Levenstein. AS/400 64-bit PowerPC-compatible processor implementation. *ICCD*, 1994.

[BRG+89] D. Black, R. Rashid, D. Golub, C. Hill, and R. Baron. Translation lookaside buffer consistency: A software approach. *Proc. of the 3rd International Conference on Architectural Support for Programming Languages Operating Systems*, pages 113–122, 1989.

[CBJ92] J. B. Chen, A. Borg, and N. P. Jouppi. A simulation based study of TLB performance. *Proc. of the 19th Annual International Symposium on Computer Architecture*, 19(2):114–123, May 1992.

[CCH+87] F. Chow, S. Correll, M. Himelstein, E. Killian, and L. Weber. How many addressing modes are enough. *Proc. of the 2nd International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 117–121, October 1987.

[Che87] R. Cheng. Virtual address caches in UNIX. *Proc. of the Summer 1987 USENIX Technical Conference*, pages 217–224, 1987.

[CK92] T. Chiueh and R. H. Katz. Eliminating the address translation bottleneck for physical address cache. *Proc. of the 5th International Symposium on Architectural Support for Programming Languages and Operating Systems*, 27(9):137–148, October 1992.

[CMMP95] T. Conte, K. Menezes, P. Mills, and B. Patel. Optimization of instruction fetch mechanisms for high issue rates. *Proc. of the 22nd Annual International Symposium on Computer Architecture*, 23(2):333–344, June 1995.

[EV93] R. J. Eickemeyer and S. Vassiliadis. A load-instruction unit for pipelined processors. *IBM J. Res. Develop.*, 37(4):547–564, July 1993.

[Gwe95] L. Gwennap. Hal reveals multichip SPARC processor. *Microprocessor Report*, 9(3):1–11, March 1995.

[Hea86] M. Hill and et al. Design decisions in SPUR. *IEEE Computer*, 19(11):8–22, November 1986.

[HHL+90] K. Hua, A. Hunt, L. Liu, J-K. Peir, D. Pruett, and J. Temple. Early resolution of address translation in cache design. *Proc. of the 1990 IEEE International Conference on Computer Design*, pages 408–412, September 1990.

[HP90] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Mateo, CA, 1990.

[Jol91] R. Jolly. A 9-ns 1.4 gigabyte/s, 17-ported CMOS register file. *IEEE J. of Solid-State Circuits*, 25:1407–1412, October 1991.

[JW94] N. P. Jouppi and S. J.E. Wilton. Tradeoffs in two-level on-chip caching. *Proc. of the 21st Annual International Symposium on Computer Architecture*, 22(2):34–45, April 1994.

[KCE92] E. J. Koldinger, J. S. Chase, and S. J. Eggers. Architectural support for single address space operating systems. *Proc. of the 5th International Symposium on Architectural Support for Programming Languages and Operating Systems*, 27(9):175–186, October 1992.

[KH92] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[KJLH89] R. E. Kessler, R. Jooss, A. Lebeck, and M. D. Hill. Inexpensive implementations of set-associativity. *Proc. of the 16th Annual International Symposium on Computer Architecture*, 17(3):131–139, 1989.

[LE89] H. Levy and R. Eckhouse. *Computer Programming and Architecture, The VAX*. Digital Press, 1989.

[LS94] A. Lebeck and G. Sohi. Request combining in multiprocessors with arbitrary interconnection networks. *IEEE TPDS*, November 1994.

[Rau91] B. R. Rau. Pseudo-randomly interleaved memory. *Proc. of the 18th Annual International Symposium on Computer Architecture*, 19(3):74–83, May 1991.

[SF91] G. S. Sohi and M. Franklin. High-bandwidth data memory systems for superscalar processors. *Proc. of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, April 1991.

[TH94] M. Talluri and M. D. Hill. Surpassing the TLB performance of superpages with less operating system support. *Proc. of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, 29(11):171–182, November 1994.

[WBL89] W.-H. Wang, J.-L. Baer, and H. M. Levy. Organization and performance of a two-level virtual-real cache hierarchy. *Proc. of the 16th Annual International Symposium on Computer Architecture*, 17(3):140–148, May 1989.

[WE88] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective*. Addison–Wesley Publishing, 1988.

[YP93] T.-Y. Yeh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. *Proc. of the 20th Annual International Symposium on Computer Architecture*, pages 257–266, May 1993. *Computer Architecture News,* 21(2), May 1993.