HEWLETT PACKARD

A Comparison of *Protection Lookaside Buffers* and the PA-RISC Protection Architecture

John Wilkes, Bart Sears
Computer Systems Laboratory
HPL–92–55
March 1992

processor architecture, single address space, 64-bit addressing, protection, PA-RISC, protection lookaside buffers, PLB, translation lookaside buffers, TLB, cache indexing, virtually-addressed caches

Eric Koldinger and others at the University of Washington Department of Computer Science have proposed a new model for memory protection in single-address-space architectures. This paper compares the Washington proposal with what already exists in PA-RISC, and suggests some incremental improvements to the latter that would provide most of the benefits of the former.

1 Introduction

A recent technical report by Eric Koldinger and others [Koldinger91] proposes a new model for managing protection information in single address space processors. This paper is one outcome of discussions with the authors about the real differences between their proposal and the scheme used in PA-RISC [Lee89, HPPA90]: it offers a restatement of the new proposal, a comparison with the existing PA-RISC architecture, and some thoughts on how the PA-RISC architecture might be extended in the future to provide some of the benefits that the PLB idea was trying to achieve.

The reader is assumed to be familiar with modern TLB, cache and processor architectures; with luck, sufficient information is provided in this paper that detailed prior knowledge about either the Washington proposal or PA-RISC isn't required.

Example

To help illustrate some of the finer points of the different protection models, an example based on the LRPC work of Brian Bershad is used [Bershad90] (see Figure 1). The idea here is that a client process performs a "local remote procedure call" into a protected server. To make things go really fast, the arguments and results are passed from client to server and back again through a shared memory area, accessible to both. The interesting case happens when the server must not be able to access any client state other than that explicitly passed in. To achieve this, the arguments and results are put in a protected portion of memory accessible to both the client and server. Other than that, the two protection domains are disjoint.

We will consider two cases: in the first, the client process is suspended at the point of the call, and a separate server process is used to service the request. In the second, a single process starts in the client domain, enters the server protection domain (without a full context switch), performs the work, and then returns to the client domain.

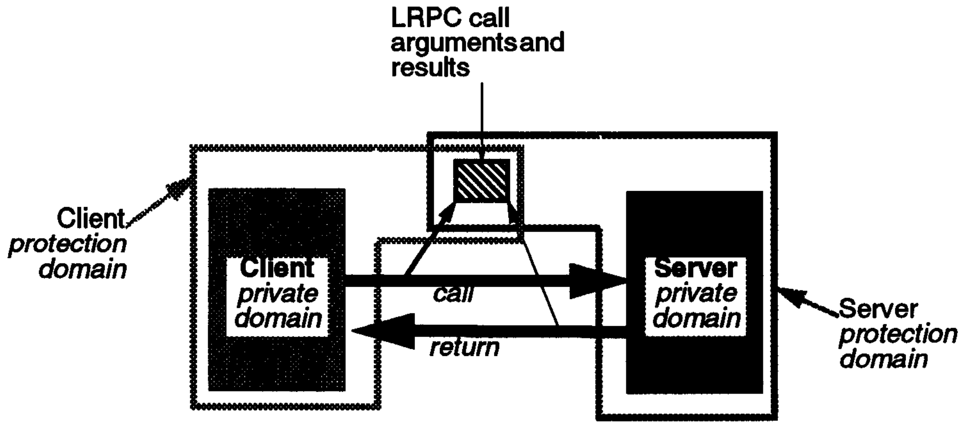

Figure 1. LRPC and protection domains.

Terminology

Here is some terminology that will prove useful later:

- *Process*: anything that executes in a processor, without regard to whether it is a task, thread, or whatever.

- *Principal*: the entity whose access rights we wish to describe. Examples: a Mach task, an HP-UX process; a database server. (In the bad old days before PA-RISC, these were equivalent to address spaces.) Each principal will be represented by one or more processes.
- *Protection set*: a set of pages with exactly the same list of valid accessors. (Examples: the stack of an HP-UX process; the protected data space of a service like a database; a shared executable code module.)
- *Sharing set*: a protection set being used for shared memory communication between two or more principles (e.g., an LRPC argument-passing area).
- *Private domain*: the protection sets available to a single principal and intended solely for its own use (e.g., its private stack and heap areas).
- *Protection domain*: the union of the private domain and the sharing sets available to an executing principal at any given moment. (The size of the protection domain is typically changed at privilege or protection domain boundary crossings.)

The rest of this paper is organized as follows. The next two sections discuss the two approaches: the Washington PLB proposal and the existing PA-RISC architecture. Section 4 offers a detailed analysis of the arguments of the former, and section 5 offers a more broadly based comparison of the two different protection models that the implementations are representing. Section 6 suggests some changes that could be made to PA-RISC architecture to make it more flexible, and section 7 offers some conclusions.

2 The protection lookaside buffer proposal

This section attempts to summarize the argument of [Koldinger91], otherwise known as "the PLB paper".

Single-address-space systems are the Right Thing To Do, and thus the wave of the future—especially with the coming migration to 64-bit virtual addresses. They solve all sorts of nasty problems (such as removing the need for homonyms and synonyms), while allowing virtually-addressed caches to be used, which results in faster processor cycle times. In such systems limiting addressability cannot be used to secure protection: every address in the system can be quoted by any principal (or process), so a separate control mechanism is needed to prevent unauthorized data accesses.

The PLB paper argues that support for multiple different simultaneous protection rights for each page is a desirable thing. Merely being able to support both read and read-write access to a single page (as can be done by PA-RISC) is insufficient: it should be possible to support arbitrary, concurrent, access types for different processes using a single page.

The basic question then arises of how best to represent this information. Clearly it needs to be held in fast memory on the processor chip: the access rights need to be checked on every cache access. The solution proposed is an associatively-searched table of protection data, indexed by a combination of the virtual page number and a protection domain identifier (or *PD-ID*). This can be achieved by using an on-chip *Translation Lookaside Buffer*, or *TLB*. The TLB contains protection and translation information, indexed by the virtual address of the page being referenced. In the PLB paper, the TLB is augmented by including the PD-ID in the index that determines which TLB entry to use.

The paper then makes a further assertion (almost unrelated to the first): given a virtually-addressed cache, you don't need the virtual to physical address translation information on-chip. Thus the on-chip TLB can be shrunk to hold only protection information: it becomes a *protection-*

lookaside buffer, or *PLB*. With a virtually-addressed cache, the PLB and cache are searched simultaneously. Translation information is held in an off-chip TLB. The paper suggests that removing the translation information from the TLB will reduce the chip area required for its implementation substantially.

When a cache miss or write-back does occur, a large off-chip TLB can be searched. The paper asserts that this off-chip TLB is not on the critical path since it is not searched when the data is present in the on-chip cache. When a PLB miss occurs, it can be reloaded using the usual TLB replacement algorithms.

Overall, the result of the new proposal will be (the paper claims) a much more flexible protection scheme, requiring less silicon area to implement, and capable of running at least as fast as a regular TLB system.

Example

The LRPC model would be implemented by inserting two entries in the PLB for each page in the argument/result sharing set: one entry for each protection domain. In the two-process case, the client process would have the client PD-ID, the server process the server PD-ID. The single-process case would be implemented by switching the PD-ID value at the protection domain crossings. Both schemes have low execution costs, although they chew up PLB slots at double the desirable rate.

3 The PA-RISC model

Since our comparison point for this evaluation is the already-existing PA-RISC architecture, we offer a brief description of the relevant portions of it here. Full details can be found in the PA-RISC architecture and instruction set reference manual [HPPA90].

The processor has four privilege levels, numbered 0 (most privileged) through 3. A process executes at one of these privilege levels at any one time; increases in privilege are effected by **Gateway** instructions, decreases by several of the **Branch** instructions.

Each page in a PA-RISC machine has a 7-bit encoded representation of the access rights to it (such as read, write, execute, and gateway[1]), and the privilege levels that can validly issue such accesses. Three bits are used to encode the access types allowed, and a further two pairs of two bits to identify the privilege levels. (For example, a single page may be marked writable from privilege level 0, readable from privilege levels 0, 1, and 2, and inaccessible from privilege level 3.)

In addition, each page also has a single protection key, known as an *access identifier*, or *AID*. The AIDs are used to represent protection sets. The processor keeps up to four of these key values in fast, protected, control registers (only code executing at privilege level 0 can load them). In the processor they are known as *protection identifiers*, or *PIDs*, because they have an additional bit to mean "disable write accesses". This means that the common case of read-only and read-write access to a single page from different processes can be handled by a single AID value, with only a single TLB entry.

An AID is currently architected to be 15 bits, so a PID is 16 bits long.

Per-page protection information is made known to the processor by loading it, together with address translation data, into the processor's TLB. An access to memory (or a cache line) is only

1. Called "promote to privilege level N" pages in the architecture reference manual.

allowed to proceed if the per-page AID matches one of the processor-stored PIDs, and the access type and privilege levels are compatible with the per-page protection data.

The union of the PIDs available to a process represents its current protection domain. If more than four different PIDs are needed, the processor will take a protection fault on a load or store instruction, and a new PID can be loaded into one of the control registers. The cost is comparable to taking a TLB miss fault—about 30 cycles, best case.

Examples

The shared LRPC argument/result pages would be implemented as a separate protection set with a unique AID. In the two-process case the related PID (with writing enabled) would be given to both the client and server processes, and cached by the processor in one of its four PID control registers. The client and server processes would each have a PID for their private domains.

The single-process case would be implemented similarly, except that the client and server private domain PIDs would be exchanged for each other at the protection boundary crossing. The execution cost is identical to the PLB proposal (no TLB flushing is needed), except that only a single TLB entry is needed for each page in the argument/result protection set.

4 Analysis

The intellectual attractiveness of associating per-page memory access rights with the process is undeniable: it is a very simple, straightforward model. Unfortunately, it is potentially expensive to implement, and the real issues arise in determining what to compromise in implementing an efficient approximation to it.

This section offers some detailed critiques of the Washington paper's arguments; the following one discusses the two underlying protection models more broadly.

4.1 Saving silicon by deleting translation information from the PLB

The PLB paper suggests that a significant amount of silicon area would be saved by removing the translation information from the TLB.[2] This seems unlikely: consider a full 64-bit virtual address, 32-bit physical address, PA-RISC level 2, TLB entry with 4096-byte pages (2^{12} bytes). It is made up as follows:

function	*size*
virtual address	52 = 64–12 bits
physical address	20 = 32–12 bits
protection data	22 = 15+7 bits
flags	4 bits
total	**98 bits**

A PLB would contain all of these fields except for the physical address. This would save only 20% of the total bit count. The silicon area saved would be a smaller percentage because of the relatively high area cost of the associative lookup logic: since most on-chip TLBs are small, they are made highly associative to achieve an adequate hit rate, which adds to the area required to implement them beyond that needed for the bit storage.[3]

[2]. The assertion is also made that first-level caches typically contain protection data. This is certainly wrong in PA-RISC, and seems unreasonable for other architectures too.

Unfortunately, the PLB paper doesn't clearly distinguish between (a) virtually-indexed (addressed) physically-tagged caches and (b) virtually-indexed virtually-tagged caches. These differences are important because they determine what information is needed when a cache line is being searched for, and, for second-level caches, what information needs to be available off the processor chip.

The processor needs to be able to do three things on a load or store: (1) do a cache lookup and test whether it got a hit—i.e., test the cache line tags of the matching lines; (2) perform a protection check; and (3), if there's a cache miss provide a physical address to the memory system. PA-RISC implementations use the physical address as a first-level cache *tag* because it is much more efficient in terms of silicon. Since the on-chip caches are typically associative rather than direct-mapped to maximize the hit rate, a cache hit on a given virtual address will produce several possible cache lines. Distinguishing between them is much better done with a tag made from a 32 bit physical address than from a 64-bit virtual one, which would take a great deal more silicon area.

Deleting the on-chip translation information in the manner proposed would require virtually-indexed, virtually-tagged first-level caches, which (as mentioned above) will be expensive: the tag and its matching logic will be larger. The result is an extra 52 bits *per cache line*—plus the extra area required for matching a 52-bit tag rather than a 20-bit one. The result will almost certainly be an increase in the total chip area needed for implementation rather than a decrease, because there are typically many more first-level cache lines than TLB/PLB slots.

4.2 Off-chip TLB performance

PA-RISC second level cache implementations can be either physically or virtually addressed since the physical address is available from the on-chip TLB. In practice, implementations so far have been physically indexed: virtual addresses are roughly twice the size of physical addresses, and this approach requires fewer pins to be driven off the processor chip.

The PLB proposal's virtually-tagged first-level cache has no on-chip physical-to-virtual translation information. Thus, if a cache line write-back is needed (e.g., a dirty line is being replaced in the cache by a new one that is being brought in), it will be necessary to go to the off-chip TLB to get the physical address of the cache line. This will require two off-chip TLB accesses: one for the cache line being loaded, a second for the one being replaced. In turn, this will increase both the complexity and the performance cost of a first-level cache miss. The alternative would be to store physical addresses with each cache line in addition to the virtual addresses needed for the tags. Avoiding the large cost of doing this is why on-chip TLBs were invented!

The best way to think about this is that having physical address data in the TLB to perform the cache line tag match also produces—for free—a convenient physical address for the memory system on a cache miss.

An assertion is made in the PLB paper asserts that the off-chip TLB is "off the critical path." Although this is strictly true of a first-level cache hit, it is not necessarily true of a first level cache miss, which is a critical component of the overall cycles-per-instruction count (CPI). In the PLB proposal, searching the off-chip TLB in parallel with the cache requires that the second level cache be virtually indexed (and possibly virtually tagged, depending on the relative speeds of the cache and TLB implementations). The alternative would be to serialize the TLB and the cache lookups,

[3] Although full associativity is not strictly necessary in the current PA-RISC architecture for correctness,it is often used to increae performance. It would become necessary on any architecture that supported more than one page size for its TLB entries.

which would roughly double the cost of a first-level cache miss—even more if there was a PLB or TLB miss. Fortunately, virtually indexing a cache is nowhere near as expensive as using a virtually-tagged one.

The effect on the CPI of an off-chip TLB needs a much more careful quantification before such a move can be shown to be a good idea. If it was found desirable, the idea could be applied to both the PLB and PA-RISC protection models.

4.3 Different simultaneous access rights

The PLB paper suggests that a great many near-arbitrary combinations of access types are going to be wanted concurrently to a single page. For example: efficient write-only access and execute-only accesses to a single page by different processes. In PA-RISC this is relatively expensive: such protection data would have to be purged from the TLB on process switches between the two processes.[4]

The PA-RISC architects obviously felt that this generality wasn't really useful, and chose instead to offer different access rights at different privilege levels, coupled with a write-disable bit in the PID. The first feature is useful for a number of things:

- efficient kernel calls: no extra PID/TLB entries are needed to represent the additional access rights of a process as it enters the kernel
- fast kernel-to-user-space communication (consider a page writable by the kernel, readable by lower-privilege code; fast inter-privilege level upcalls [Clark85a]).

The second feature provides an efficient implementation for copy-on-write pages, and producer-consumer/producer-observer relationships.

Is a more general facility useful? We suggest not: the benefit isn't commensurate with the costs. (At least given the current arguments in the PLB paper—perhaps more convincing, fruitful examples could be provided. The list of proposed interesting cases could as well be handled by the PA-RISC scheme.)

> Suppose nonetheless that some good use could be found for a fully-general encoding of the access rights. This would presumably take 1 bit each to represent the access rights **read**, **write**, **execute**, and **gateway**. Preserving (for the moment) the 4 PA-RISC privilege levels would mean a total of 8 bits in the per-page access information (compared to 7), but lose the special meanings assigned to "write below level A, read below level B" currently possible. These could be recovered if an additional 4 bits (total 12) were added to allow separate representation of the access rights at the high and low privilege levels.
>
> Alternatively, you could plausibly argue that the 4 levels in the current PA-RISC architecture aren't really useful (we thought they were when we designed it, but events since then haven't convincingly demonstrated the validity of such a ring-like system.) Suppose we adopted a 2-level privilege model (levels 0 and 1). Then a new protection encoding might be: 3 bits for **read**, **execute**, **gateway**, and one bit each for **write** at the two different privilege levels. Total: 5 bits. This would allow encoding of "read-only at privilege level 1, writable at privilege level 0", which could be useful for kernel-to-user space upcalls.

[4]. Not every process switch, notice: just ones that accomplish a transition between the two different protection domains.

Taking a more aggressive tack, you could do away with all the access type information from the TLB: instead, add it to the PIDs. That is, instead of the current write-disable bit in the PID, you could encode **read, write, execute,** and **gateway** explicitly in the PID. The per-page AID would be used to identify which PID to use; different principals could be given different access rights bits in their PIDs.Note that this proposal would not change the number of bits in the AID (i.e., the number of bits in the TLB would not increase): all that is needed is some extra bits in the PID registers.

Another scheme has been suggested by Eric Koldinger. In this, *all* protection is done through the memory protection system—including determining whether sensitive operations like "load control register" can be performed. How might this be done? One possibility would be to make every potentially dangerous operation be represented as a read or write of a processor register in some portion of the processor's physical address space, and then use the protection mechanism to limit access to such pages. (There's some bootstrapping problems about loading the TLB/PLB here!)

A perhaps better way to implement this last idea would be to encode an additional "execute privileged" access right on a page: if the processor was executing from such a page, it would be allowed to issue the sensitive instructions. This would then remove the need for privilege levels, and require only 5 bits of protection per page. It would of course sacrifice the PA-RISC encoding of different access rights at different privilege levels.

Since the variance in these schemes represent only 2–5% of the TLB entry size we can perhaps agree to ignore them as a matter of taste for this discussion.

4.4 Identifying protection sets

The PLB paper argues that the use of (protected) sharing between processes will greatly increase in the future. This may very well be right.

For example, the high performance LRPC system of Brian Bershad [Bershad90] requires a protected area used for argument passing that is accessible only to a client and the protected server it is accessing. Such communication can be pairwise or *N*-way, depending on the intent. In such cases, the principles need a handle to represent their ability to access these pages: in short, one or more keys held by the process and used to test against the PLB or TLB entries.

How many different keys will be needed? In the PLB model the PD-ID is used as the key to distinguish each different protection domain, so that they can be given different access rights. These distinguishing identifiers are loaded into the processor when a context switch occurs; each process has exactly one PD-ID. The PD-ID has to be long enough to represent the maximum number of different protection domains in the machine. To a first approximation, this will equal the count of Mach-like tasks or HP-UX-like processes in the system, plus a small number for the protected services available. Something in the range of 15–20 bits seems reasonable. (Imagine a 1 GIPS processor, executing TPC-A at roughly 20 000 transactions per second (tps). The tps count is about a third of the process count for the most naive implementation, so this would need at least a 16 bit PD-ID.)

In PA-RISC, the number of potential PIDs is going to be larger: rather than being about the same as the number of processes, it is equal to the number of processes plus the number of protected shared-memory communication interactions they all establish simultaneously. (Because of the privilege levels, the user–kernel interactions don't need to be handled via PID allocation.) So: each process needs one PID for its private data, plus a PID for its executable code. Let's agree to do aggressive code-sharing (this is a single address-space machine, after all), so one PID can be used to protect all the read-only code in the machine. Suppose, for the sake of argument, that an average process indulges in 3 additional protected interactions—e.g., with the file system, the database, and a network server. The average process then needs 5 separate PIDs, of which 4 are private to it. In turn, this suggests that there will be about 4 times as many PIDs needed as PD-IDs, so a PID will take 2 more bits than an PD-ID to represent.

Note that the PA-RISC architecture is optimized for up to four sharing groups per process: if more are needed, then they can be reloaded on demand with a cost comparable to handling a TLB miss.

This analysis suggests that the PLB protection key will be about 2 bits shorter than a PA-RISC AID. This is about 2% of the total TLB stored bits. However, this may not represent a saving of silicon area: whereas the (highly associative) TLB lookup and match is done on the 52-bit virtual page number, the PLB lookup and match algorithm has to use the 52-bit page number plus the 15–20 bit PD-ID, since it is effectively part of the address space for matching purposes. This may well remove any gains from storing fewer bits in the PLB.

4.5 Multiple PLB slots for a page

Of course there's a further complication in doing a direct length comparison for the PLB and TLB slots (you expected less?): the PLB proposal requires an additional PLB slot for each process using a page. Thus, a common case—read-only for one process, read-write for another, which could be represented by a single PA-RISC TLB slot—would need two slots in the PLB. Given the severe limitations on TLB slots in current machines, this is likely to greatly increase the PLB miss rate over a comparable TLB (e.g., the new DEC Alpha chip has only 8 I-TLB and 32 D-TLB entries). Any proposal to move to a PLB should include a very careful analysis of this cost: it's a high price to pay for a small increment in convenience!

A minor point: the semantics of PLB flush/purge operations will need some thought. Should they discard all slots for a given address (this may not be feasible in the memory technology)? Or should they just delete those mappings for a single specified address/PD-ID pair (which may make it hard to get rid of them all)?

5 An overview of the two underlying protection models

The previous sections compared the PLB paper and the PA-RISC architecture in some detail, concentrating on the first-order differences. This section attempts to pull back a bit from the two models and consider the pair of overall protection models that they are implementing.

First some background. One approach to protection, of long standing, is to use "base&bounds" registers to limit the addresses that can be issued by an executing process. This technique is typically used in processors without virtual memory. For example, the new Inmos T9000 transputer [Inmos91] has a single machine-wide (non-virtual) address space together with four base&bounds registers per process (i.e., in processor control registers) to limit access to subsets of the address space. This technique isn't commonly used in a machine with virtual addressing because it makes paging hard to implement. (There are occasional exceptions: for example, the PA-RISC 1.1 architecture has provision for a variant of the base&bounds approach to associate access information with virtually-contiguous segments that are integer multiples of the page size.)

The current popular alternative is to use a page-based protection scheme. The simplest technique is to associate a process identifier with each page, and change this as needed. This is obviously inadequate: it makes it hard for different processes to share data—each context switch would produce a flurry of protection data changes on shared pages. Instead, some mechanism for representing grouping of access right is used. The two techniques being discussed here are, briefly:

- Associating access rights with the page, as in PA-RISC. Only the right principals can access a given page: an *access control list* (or *ACL*) model.

  The AID identifies the set of processes that may access a page; pages with different sets of accessors have to have different AIDs. An AID is a unique name for each such ACL.[5]

  Implementing these ACLs efficiently is slightly complicated in PA-RISC because there are relatively few bits in current AIDs, so they have to be conserved and reused as much as possible. In particular, the OS can't simply allocate a new one each time it discovers it has more than one valid accessor for a page. The limited number of cached PIDs (currently 4) also suggests that the number of PIDs allocated to any one process be kept small to avoid protection-fault thrashing.
- Associating access rights with the principal, as in the PLB paper. The principal is given a list of pages that it has rights to, in the form of PLB entries: a *capability model.*

6 Extending PA-RISC

How might the PA-RISC model be extended?

It would be easy to lengthen a PID to 20 bits (there are some reserved bits for this in the appropriate places); further extensions to longer PIDs would require changing the format of the InsertTLBprotection instructions. This would ease some of the pressures on reusing PIDs because of their relative scarcity. The next thing to do would be to decouple the architectural visibility of PID registers in the processor from their exact number—just as the number of TLB or cache slots isn't architected. Here's one way to do that.

Provide a fully-associative set of PID slots that can be loaded via an insert-PID style interface.[6] These slots would be anonymous, unlike the current PA-RISC scheme, where they are explicitly named as separate control registers. The processor hardware would manage these slots in a least-recently-accessed fashion: if an insert-PID operation was performed and all the slots were used, the slot that had last been used for a successful permission check would be discarded. The number of PID slots can now be implementation dependent; an architectural minimum of 2 may prove to be healthy. Future machine implementers can decide to implement more (or fewer!) slots than the current four, and this would be completely transparent to the operating system, after a one-time architecture change.

Adding a PID to a processes' protection domain will be as simple performing an "insert PID" operation. Deleting PIDs could be done in one of three ways:

- purge by slot number—good for deleting all the PIDs with a few instructions;
- purge by PID value—good for removing a particular protection set from a process;
- purge all PIDs—this sounds drastic, but is probably easy to implement in hardware, and will have exactly the right effect on protection domain crossings: all the old values would be discarded in a single cycle, and new ones could be loaded in as needed. The code and private domain PIDs would be loaded immediately, the rest could either be loaded immediately or in an as-needed (lazy) fashion on subsequent accesses. This is probably at least as cheap in

5. A slightly confusing attribute of the implementation is it appears to be almost the opposite of the model it is implementing: PA-RISC PIDs look like capabilities when they are really access control list names.

6. The obvious name for this is of course the *Protection Lookaside Buffer* ....

cycle count as a scheme that used fancy indexing algorithms to keep lists of PIDs that needed to be purged. We recommend this approach.

Now that the functions are clear, we still need to determine how to express them to the processor. At this point, the conveniences of modifying processor state through control registers or new instructions are going to be the determining factor, and we have no feeling one way or the other. The two schemes we have thought of are:

- Define a new LoadPID instruction to insert a PID into the stack, by analogy with the way that TLB entries are loaded. Similarly, a PurgePID operation would remove all of the stored PIDs from the stack.
- To avoid introducing new instructions, assign a single control register as an interface to the PID slots. Storing a value into this register would insert the new PID into the set (possibly displacing one already there). Writing a special PID value such as zero would delete all the entries in the stack.

The interface to the PID slot set is clearly modelled closely on the operation of a regular TLB. Rather than (as in the PLB proposal) separating addressing information from protection data, this approach provides separate hardware support for per-process (PID) and per-page (TLB) access controls—but in such a way as to minimize the amount of additional silicon required for its implementation.

If it can be shown that there are interesting/useful protection types on a single page other than the encodings currently provided for in PA-RISC, these could cost-effectively be implemented by adding additional access type bits to the PID values.

7 Conclusions

This paper has provided an analysis of the PLB and PA-RISC protection implementation schemes. We showed that many of the facilities considered desirous in the PLB paper are already in PA-RISC—in some cases in a more efficient form than the PLB proposal.

Although our analysis favours the basic PA-RISC approach, the PLB paper raised some useful questions about possible limitations in the PA-RISC scheme. In particular, the current limitation of only 4 processor PID control registers is a potential performance hit if many more sharing sets become the programming paradigm of choice—and there are indications that this is happening with the advent of microkernel-based systems.

This analysis led to some suggestions for enhancing the PA-RISC model to increase flexibility and provide greater performance in an architecturally-transparent fashion—but these are relatively minor extensions to an already gratifyingly strong and flexible architecture.

Acknowledgments

This paper couldn't have been written without active and enthusiastic discussions with Eric Koldinger, Hank Levy, Jeff Chase, Ed Lazowska and Susan Eggers. The opportunity to have these interactions came about as a result of the University of Washington industrial affiliates program. Jerry Huck, Brian Bershad, and Dale Morris also provided valuable inputs on earlier drafts.

References

[Bershad90] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. Published as *ACM Transactions on Computer Systems* **8**(1):37–55, February 1990.

[Clark85a] David D. Clark. The structuring of systems using upcalls. *Proceedings of 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Washington). Published as *Operating Systems Review* **19**(5):171–80, December 1985.

[HPPA90] Hewlett-Packard Company. *PA-RISC 1.1 architecture and instruction set: reference manual,* Part number 09740–90039, November 1990.

[Inmos91] SGS-Thomson Microelectronics Group. *The T9000 transputer products overview manual,* 1st Edition, 1991.

[Koldinger91] Eric J. Koldinger, Henry M. Levy, Jeffrey S. Chase, and Susan J. Eggers. *The protection lookaside buffer: efficient protection for single address-space computers.* Technical report 91–11–05. Department of Computer Science and Engineering, University of Washington, November 1991.

[Lee89] Ruby B. Lee. Precision Architecture. *IEEE Computer* **22**(1):78–91, January 1989.