Report Number: WUCSE-2007-43

2007

Implementing Legba: Fine-Grained Memory Protection

Sheffield, Sowell, and Wilson

Fine-grained hardware protection could provide a powerful and effective means for isolating untrusted code. However, previous techniques for providing fine-grained protection in hardware have lead to poor performance. Legba has been proposed as a new caching architecture, designed to reduce the granularity of protection, without slowing down the processor. Unfortunately, the designers of Legba have not attempted an implementation. Instead, all of their analysis is based purely on simulations. We present an implementation of the Legba design on a MIPS Core Processor, along with an analysis of our observations and results.
... Read complete abstract on page 2.

Implementing Legba: Fine-Grained Memory Protection

Sheffield, Sowell, and Wilson

Complete Abstract:

Fine-grained hardware protection could provide a powerful and effective means for isolating untrusted code. However, previous techniques for providing fine-grained protection in hardware have lead to poor performance. Legba has been proposed as a new caching architecture, designed to reduce the granularity of protection, without slowing down the processor. Unfortunately, the designers of Legba have not attempted an implementation. Instead, all of their analysis is based purely on simulations. We present an implementation of the Legba design on a MIPS Core Processor, along with an analysis of our observations and results.

2007-43

Implementing Legba: Fine-Grained Memory Protection

Authors: Sheffield, Sowell, Wilson

Corresponding Author: mlw2@arl.wustl.edu

Abstract: Fine-grained hardware protection could provide a powerful and effective means for isolating untrusted code. However, previous techniques for providing fine-grained protection in hardware have lead to poor performance. Legba has been proposed as a new caching architecture, designed to reduce the granularity of protection, without slowing down the processor. Unfortunately, the designers of Legba have not attempted an implementation. Instead, all of their analysis is based purely on simulations. We present an implementation of the Legba design on a MIPS Core Processor, along with an analysis of our observations and results.

Type of Report: Other

Implementing Legba: Fine-Grained Memory Protection

Aaron Sheffield, Ross T. Sowell, Mike Wilson
Department of Computer Science & Engineering
Washington University in St. Louis
One Brookings Drive, Campus Box 1045
St. Louis, Missouri 63130-4899
{ajs6, mlw2}@cec.wustl.edu
rsowell@gmail.com

***Abstract*—Fine-grained hardware protection could provide a powerful and effective means for isolating untrusted code. However, previous techniques for providing fine-grained protection in hardware have lead to poor performance. Legba has been proposed as a new caching architecture, designed to reduce the granularity of protection, without slowing down the processor. Unfortunately, the designers of Legba have not attempted an implementation. Instead, all of their analysis is based purely on simulations. We present an implementation of the Legba design on a MIPS Core Processor, along with an analysis of our observations and results.**

Keywords: Legba, protection, PLB

I. INTRODUCTION

Protection is a long-standing problem in operating systems safety. With the growing popularity of mobile code, the proliferation of third-party operating systems extensions, and the clear dangers of running these extensions in a privileged context, there is a definite need for better protection mechanisms.

Recent work on Microsoft's Singularity[8] project rely on software isolated processes to provide safety properties. Type-safe languages do provide strong software protection mechanisms for safety. However, given the frequency of defects in software systems resulting in vulnerabilities, we suggest that some additional layers of protection may improve the situation.

Brian Bershad observed that while software protection mechanisms provide the most flexible and applicable protections, "software mechanisms usually rely on hardware as a foundation to ensure their own integrity, while changes in hardware protection are usually controlled and limited through software mechanisms." [10] We agree whole-heartedly with this observation, and believe that hardware mechanisms should be designed and evaluated for providing fine-grained encapsulation of light-weight objects.

Legba[2] is a hardware design for fast, fine-grained memory protection. Unfortunately, the original designers have not yet attempted an implementation. While the design appears to provide the protection mechanisms required, the design needs validation beyond simulation. We have attempted to implement Legba in VHDL to better understand the design space and to validate the Legba achitecture.

II. RELATED WORK

The problem of providing fine-grained memory protection in an efficient manner is not new. Most current processors provide a partial solution by attaching permissions to pages in the Translation Look-aside Buffer (TLB).

This has two disadvantages. The lines in a TLB refer to pages. Thus, permissions can only be as fine-grained as the page size. For most processors, this is 4KB. Secondly, the TLB is shared among all processes on the system. We must either add context tags to the TLB lines, or perform a complete TLB flush on a context switch.

This solution is also usually limited in the number of entries. Because the TLB is frequently fully associative and must be on the processor core (and critical path), the size of the TLB is minimized to reduce the access time and avoid lengthening the processor critical path.

Given the inadequacy of current processor architectures for providing fine-grained protection, research efforts have been made in new architectures for protection.

A. Software Solutions

A number of software solutions have been proposed to provide the flexible protections required. While all of these are useful in their own space, they still rely on hardware for a foundation.

One method of safely running untrusted code is to use an interpreter. If implemented correctly, this does provide complete safety. However, interpreted code is slow and inefficient. Some studies have found that interpreted code is up to 100x slower than native machine code. [13] Finally, while a correct interpreter provides complete safety, proving the correctness of a large piece of software such as an interpreter is usually challenging.

Proof-Carrying Code [14] is a much more promising line of research. A proof of safety is embedded in a program. Before

loading the program, the system checks the proof against the code and determines whether it is safe to run. This provides the best of both worlds: the mathematically demonstrated safety with the speed of native code. Unfortunately, creating these proofs has proved to be a very difficult problem, and an automated safety prover is still out of reach.

Type-safe languages [15,9,8] provide another approach. The language does not provide constructs for violating the type-safety of object encapsulations. Each component is software isolated from every other object. The major problem here is in ensuring that each program is from a type-safe compiler, and that they each adhere to the software protection scheme. Additionally, there is the non-trivial problem of validating the correctness of a large, complex compiler.

Software Fault Isolation isolates faults in one module from impacting another module. One very effective method is NOOKS, outlined in [16]. NOOKS uses a combination of automatically instrumented code and memory management techniques to prevent a defective kernel module from corrupting the rest of the kernel. However, the memory management techniques rely on modifying the TLB to restrict access to memory. While this prevents a defective module from inadvertently trampling kernel memory by mistake, no attempt is made to prevent the module from loading a new TLB and accessing at will. In the words of the NOOKS designers, "We trust kernel extensions not to be malicious, but we do not trust them not to be buggy." This is a valuable design space, but we are interested in assurance from malicious code, not merely defective code.

B. Palladium

Palladium [17] attempts to solve the same problem using existing segmentation and paging hardware in the Intel x86 class of processors. Since these processors support a large (8,192) number of segments, each with its own privilege level, as well as near arbitrary size, this seems like a promising option at first. Unfortunately, at closer examination there are some significant disadvantages.

These segment protection levels are limited to 2 bits, or four ordered levels. This is adequate for a very shallow ordered hierarchy. However, for an arbitrary protection matrix, we need to be able to support non-hierarchical orderings of arbitrary depth. In short, a complete subject/object access control matrix [18].

Palladium also has a significant protection domain boundary crossing penalty of 142 cycles. While this is not an insurmountable difficulty, it is possible to reduce the protection domain boundary crossing penalty.

C. Itanium

Itanium provides a Protection Key Register (PKR) [19]. This PKR is a 16-entry fully-associative cache. Since the cache lines do not have context tags, the cache must be flushed on every access.

Since the PKR is separated from the TLB, TLB cache lines can be shared between different protection domains, with different access permissions. However, since the PKR still ties protection to individual pages, we have no sub-page protection. Furthermore, the PKR is on the processor critical path. This is probably the driving force behind the choice of PKR size; a large, fully associative cache would add significantly to the processor critical path length.

D. Protection Look-aside Buffers

A major innovation in protection management comes with protection look-aside buffers (PLBs) [4]. With PLBs, we remove all protection information from the TLB and place it in a new construct, the PLB.

If we use virtually addressed L1 cache, then the TLB is no longer necessary for L1 cache operations, and can be moved off core. This allows us to expand the TLB and increase the associativity, since it is only used during L2 or lower accesses; the latency can be masked in the lower level memory access.

Another major benefit is that the PLB is much smaller than the TLB. Unfortunately, the PLB still suffers from all of the classical drawbacks of a TLB. It is still fully associative, and still sits on the processor critical path.

Finally, the classical PLB suffers one more major disadvantage: the need to perform an associative lookup of an address without knowing the base address of the object whose attributes are cached. Thus, when we wish to check the permissions associated with an address, we need to use the address to look up the associated memory object in a fully-associative cache using object base and limit. Having this lookup in the L1 cache is expensive.

E. Mondrian Memory Protection

Mondrian Memory Protection [7] incorporates the concept of the PLB and adds an additional optimization, the *sidecar*. Sidecars are cached protection bits applied to any register capable of containing an address (including the program counter).

When first a register is used to address memory, the memory permissions are not known. However, once the permissions are known, they are cached in a sidecar to the register. Future accesses check the sidecar first; if the address object is still the same, the sidecar permissions are used instead of consulting the PLB for permissions. This removes the PLB from the processor critical path.

In the case of mondrian memory protection, the sidecar contains:

- the base of the last memory access
- the limit of the last memory access
- the rights of the last memory access

As no context tags are included, the sidecars must be flushed on each context tag. PLB entries *do* include context, allowing us to share cache lines between different protection domains using different permissions.

Mondrian memory protection still uses a classical PLB, and suffers from the major disadvantage of the PLB: the associative lookup.

III. Overview of Legba

Legba provides fine-grained memory protection by combining features of many disparate memory protection schemes.

Mondrian memory protection comes closest to meeting our needs, except for the associative lookup. That is, we have an address, A, which falls between some object O's base B and limit B+L. Then we should apply the permissions of object O to accesses to address A. The problem becomes: how do we associate address A with object O? In mondrian, the address is checked in L1 cache.

In the case of Legba, we no longer store address objects in the PLB – in L1 cache – by base and limit, but by Object Identifiers (OIDs). This also allows us to add a level of indirection from the data cache lookup to the actual protection information, permitting us to share cache lines between multiple protection domains.

To add this indirection, Legba supplies a data cache with the usual data, tags, and stats fields; as well as an OID used to index a second cache, the Protection Key Cache (PKC), with protection information. To avoid extending the processor critical path, the PKC is placed in the pipeline stage following the data cache.

To facilitate the sharing of data cache lines between different protection domains, the PKC is indexed by *both* the OID from the cache and the current Protection Domain Identifier (PDID). See Figure 1 [2].

Legba uses four permission bits: e(X)ecute, (S)witch, (R)ead, and (W)rite. X, R, and W are self-explanatory; S is iscussed in more detail under the new instructions for Legba.

Legba also includes sidecars *à la* mondrian memory protection. However, in the case of Legba, the sidecars do not contain the full base+limit, but just an OID. During data lookup in the cache, an authoritative OID is returned. Sidecar content is validated by comparing OIDs.

Finally, Legba also adds two new instructions: Branch-Linked-Locked (BLL) and Switch-Load (SL), for managing the current PDID.

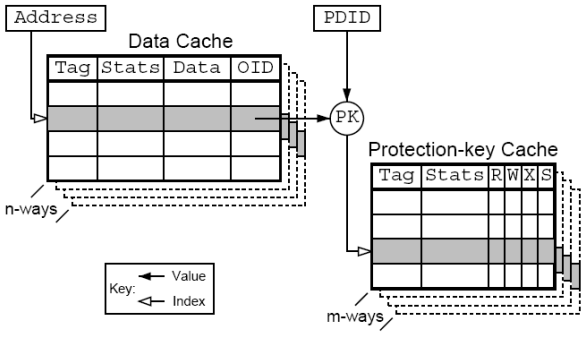

Figure 1: Cache and PKC relationship
(Picture copyright A. Wiggins [2])

IV. Implementation

Our implementation is based on the MiniMIPS [11] implementation of the MIPS core. The MiniMIPS is a standard 5-stage (IF|ID|EX|MA|WB) pipelined architecture with support for exceptions, pipeline stalls on branches, and no cache or virtual memory (and thus no TLB). We found that Legba adapted very well to implementation on the MiniMIPS.

In our implementation, an attempt to access memory in violation of policy generates an exception. In the MiniMIPS, an exception is handled by an immediate jump to a global exception handling address. Our implementation does not actually include the exception handler, which we consider to be outside our scope.

The pipeline stall on branch required one minor change to our BLL/SL instruction implementation, detailed below.

The lack of a TLB means that we cannot mask the latency of the associative lookup of the OID by performing this in parallel with address translation. Since most processor architectures today *do* have a TLB, we consider this to be an unusual case that does not invalidate the assumption that the latency can be hidden with a parallel lookup.

In general, we believe that the Legba architecture can be added to almost any processor architecture without great difficulty, although we only offer the anecdotal evidence of our own implementation as proof.

A. Pipeline Changes

Our implementation made very few changes to the pipeline structure.

We added several new components to the pipeline. The major addition is the Protection Key Cache, or PKC. See Figure 2 [2]. Also, since the MiniMIPS had no cache, we also added the instruction cache and data cache to the pipeline, for a modified Harvard architecture. In most architectures, this is unnecessary, since a separate instruction and data cache are commonly included.

We also found it necessary to add some additional registers to the pipeline. We added a PDID (Protection Domain ID), to represent the current execution context, and a Protection Key

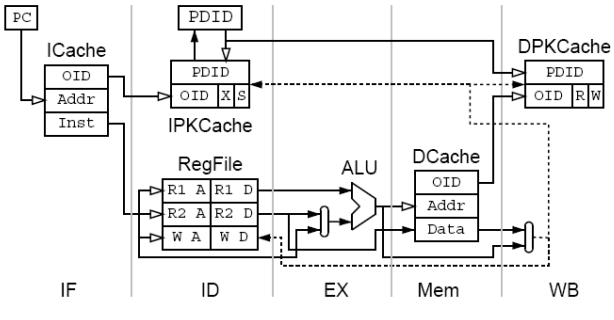

Figure 2: PKC location within Pipeline
(Picture copyright A. Wiggins [2])

Table Register, which will be discussed under the memory hierarchy.

Finally, it is worth detailing exceptions in the MiniMIPS. When an exception is raised, all stages prior to the exception are filled with NOPs, and a signal is sent to the PC register. The next address to be loaded in the instruction fetch stage will be the address of the exception handler. For our implementation, we used address 0xFFFF0000.

B. Instructions Added

Legba requires the addition of two new instructions: BRANCH-LINKED-LOCKED (BLL) and SWITCH-LOAD (SL). To simplify our implementation, we changed the BLL to a simple JUMP instead of a BRANCH-LINKED. That is, we do not save the old address on a stack, etc. Instead, we treat the BLL much more like a syscall or trap instruction. The exact semantics are as follows:

On reading a Branch-Linked-Locked, the processor sets up for a normal JMP, passing the address to the MiniMIPS jump logic. The jump logic handles relative branch computation and stalling the pipeline.

We also add an additional pipelined signal which passes through to the Instruction Fetch stage with the jump information. This signal notifies the IF stage that the next instruction must be SL.

On load, we have additional logic we have added to the IF stage that checks the signal and the next instruction. If a BLL is not followed by SL, an exception is raised.

Before executing the SL instruction, we verify that we have (S)witch permission to the object in which it resides. When executed, SL takes the OID of the object containing the SL and loads it into the pipelined PDID register.

The actual validation of the X and S permissions are addressed in our discussion of the PKC component. It is sufficient to note for now that we must have X permission to the object in which both the BLL and SL reside, and S permission for the object in which the SL resides.

One point not discussed in the original design of Legba is whether a naked SL should generate a permissions exception. We elected to allow this case. Thus, SL may be encountered in any point in the program. However, given that the only way this can happen is if we "fall through" from one object space to another, or by jumping to an SL by a normal JMP (where a BLL could be used), this seems to be mostly irrelevant.

In summary: our implementation adds two instructions to the MiniMIPS instruction table, the BLL and SL. We also add logic to the IF stage to verify that SL follows BLL (component `is_SL`). Finally, we add logic to the ID stage to load the new PDID into the pipeline (component `pdid_update`). Since the PDID is only used in the ID and WB stages, we latch it in stage ID as a register, and pipeline it through to the WB stage as a normal pipelined signal.

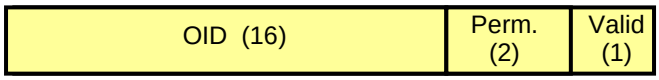

Figure 3: Sidecar contents

C. Sidecars

Legba implements Mondrian-style sidecars, but instead of a base+limit identifier, we use an OID. Because of concerns in the caching components, we made our OIDs 16 bits.

Our sidecar adds 19 bits to the storage for each register (see Figure 3). Note that we only have 2 bits of permissions, despite having 4 distinct protection bits. This is because we have two distinct types of registers. The PC is only ever used to execute or switch, never to read or write. The other general registers are only ever used to read and write, never to execute or switch. Thus, we can save 2 bits on each sidecar by only caching the relevant bits.

Our implementation has the following semantics:

1. On context switch (SL instruction), all sidecar "valid" bits are cleared.
2. When a register is used in a lookup, real permissions are validated for that lookup in the cache and PKC.
3. As a side benefit to checking these permissions, we also route them back to the register used in the lookup, as a sidecar update. This update may also change the OID of the register.

The implementation requires only a few changes to the existing MiniMIPS: add storage bits to the register, and "update" and "flush" lines to the register files. We also add pipelined signals to track which register is being used in a memory access, so that sidecar updates are routed to the correct register. This adds no logic to the top level, just an additional signal for the pipeline.

The sidecar updates take the exact same form as the sidecar data itself, where the "valid" bit is used to assert an update.

D. Memory Hierarchy

Our cache architecture is a 4-way modified Harvard architecture. (See figure 4) While the addition of two more L1 cache components runs the risk of more frequent stalls, the original designers believe that PKC misses should be rare. We discuss this assertion further in our evaluation section.

An important point not addressed in either the original Legba design or in our own implementation is cache coherence. Given that we now have 4 separate cache components in L1, we believe it would be most efficient to implement request bus snooping to maintain coherence. Additionally, bus snooping allows us to monitor changes to permissions in the lower level Protection Key Table (PKT), updating the relevant PKC and/or cache. The original Legba design also mentions the possibility of object re-numbering; this could be an acceptable way to implement this.

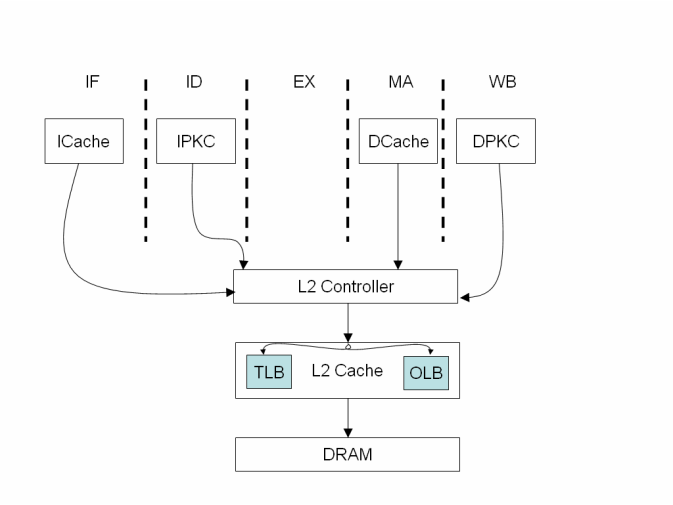

Figure 4: Legba Memory Hierarchy in the MiniMIPS architecture

In our actual implementation, we completely ignore the issue of cache coherence and recommend this to future investigators.

1) L1 cache

Our L1 cache scheme has 4 different components. However, the instruction cache and the data cache can be implemented by the same component, as can the two Protection Key Caches (PKC's).

Our actual cache was never intended to be synthesizable, and was implemented in behavioral VHDL with some delay statements. The problem of implementing cache has long been solved, and we use the behavioral VHDL to model a real cache.

Our cache model is for n-way set associative cache. Since the MiniMIPS does not have virtual memory, it is physically addressed and physically tagged. We chose to make it 4-way set associative, but do not attach significance to this number. We also add an OID to each line, but all logic for selecting this OID is in the L2 cache, not L1. Our L1 performs simple caching only.

We do encapsulate our L1 cache inside logic to handle an unexpected data dependency, explained in our evaluation section. This logic is not present in the original Legba design.

Our PKCs are significantly more complicated.

We encapsulate our actual PKC inside a set of logic to handle sidecar optimizations, pipeline stall management, and protection checking.

A PKC check requires the following inputs:

- OID (from cache),
- sidecar from register, with OID, permission bits, and validity,
- current PDID (from pipeline),
- access type request (from the instruction, e.g., Read, Write)

PKC outputs:

- sidecar updates (routed to the relevant register)
- exception line (on access violation)
- stall (on cache miss and lower-level lookup)

Our PKC also has one other output used in managing the data dependency mentioned above, detailed in our evaluation section.

On a PKC lookup, we first check the cache OID against the sidecar OID. If they match, and the sidecar is valid, we can skip the PKC lookup and used the sidecar permissions.

Otherwise, we must look in the PKC. The PKC is a set-associative cache, indexed by OID. We used the PDID as the tag in our lookups. Thus, indexing by OID returns a set of cache lines; using the PDID, we can do a parallel check of the set of cache lines, searching for the correct one. Since our lines are very small (PDID + permissions = 18 bits; adding replacement algorithm stats, a line is probably 20 bits), our parallel lookup is very fast, and we can afford to have a fairly large PKC.

In either case (sidecar or PKC lookup), the protection information is routed with the requested access information to a small protection check component (`permchk`). This component issues the actual exception to the external pipeline.

During a PKC miss, we need to look up the information from lower levels. However, the protection information is actually stored in a PKT in DRAM. This information is not stored in the same format as in our PKC. The PKT (described under DRAM) is a two-level hash table, so a lookup of protection from lower levels involves two distinct reads.

We implemented this as a 3-state FSM within the PKC. Normally, the FSM is in the idle state. On a miss, we begin reading from the first hash table (READ1). After receiving the first read, we can read from the second level (READ2). The actual read is from L2, so we may have varying stalls, based on the availability of L2 and the hit rate within L2 for the PKT.

Since the PKT lines (described under DRAM) are not in the same format as in L1 and L2 cache, a read from line will return protection information for potentially multiple protection domains. In our implementation, we discarded this information. An alternative implementation would be to store this information in adjacent lines within the PKC object set. We do not consider this particularly valuable. The line will still be cached in L2, so subsequent reads will not need to look to DRAM, and since OIDs may share sets if the PKC is small, we do not wish to evict one object's protection in favor of another.

In summary: our L1 cache is fairly standard. Our PKC is the most custom component in our implementation, and does 2 lookups on a cache miss, due to the 2-level PKT.

2) L2 cache

Since we now have a 4-way shared L2 cache, we use a simple L2 controller to arbitrate access. Our arbiter (`L2_controller`) always gives automatic priority to the

request from farthest along the pipeline. Since prior stages cannot advance even if they have the information from L2, this simplifies pipeline management.

We used this memory architecture as a shortcut. We believe that a better implementation would involve a bus, and bus snooping by each caching component to detect updates to locally cached information. However, particularly in the case of the PKC, it is difficult to see how the PKC can recognize that a change to the PKT has been made. While indexing from the PKTR can be detected, the PKT is a 2-level hash table, and detecting a modification to the second level is difficult.

As is typical, we have an address signal, a data bus for data transfer in or out, a set of input enable signals (chip, read, and write enable), and an output enable signal (data ready). Our L2 cache lines are shown in Figure 5.

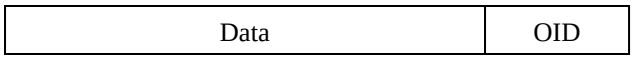

Figure 5: L2 cache line

Our L2 cache contains two components not usually found at L2: the TLB and a new component, the Object Look-aside Buffer (OLB).

Since the MiniMIPS has no virtual memory, we do not actually implement a real TLB. We did mark the point where it would exist in the L2 cache, and place a dummy "pass-through" component.

Choosing to locate the TLB in L2 cache is possible because we envision Legba being implemented in an architecture with a virtually addressed L1 cache. Since protection has been moved out of the TLB and into the PKC, there is no longer any need to keep the TLB on-core, in the L1 region.

We *do* implement the OLB. The OLB is a fully associative array of the available objects with base and limit. A lookup in the OLB takes an address and searches all object in parallel to return the OID in which the address resides. Since all cache lines within the L2 cache already contain an OID, only lookups to DRAM require searching the OLB. Thus, we mask the OLB latency behind the DRAM access latency.

The remainder of L2 cache is fairly normal, non-associative cache. Once again, we have implemented this as behavioral VHDL using delay statements for simulation. No attempt is made to make the L2 cache data portion synthesizable.

The sequence of operations in the L2 cache is as follows. Arbitration is left out of this sequence, as it should be clear without discussion.

1. A request comes in to read/write to some address, along with assertions on `c_en` and `rd_en` or `wr_en`.
    a. If the data is present, it can be read/written immediately. `d_rdy` is asserted.
2. On a miss, we perform two tasks in parallel.
    a. Index the address into the OLB, obtaining an OID for the data. (On an OLB miss, we generate an exception.)
    b. Issue the read to DRAM, waiting until DRAM is ready. Note: we assume write-back cache, so the data is read into L2 cache even on a write.
3. The data is copied into L2 cache. The stats are set, and the OID is appended to the data line.
4. Finally, any pending read/write is done, and `d_rdy` is asserted.

Note that since both L1 and L2 cache have the same cache line format, the data and OID are returned on one big bus which is redirected directly into the relevant cache line.

3) DRAM

Our DRAM is implemented in totally non-synthesizable behavioral VHDL with explicit delay statements. Our DRAM consists of a large array of words. We make no effort to do a realistic "progressive delay" in our DRAM, but just apply a constant penalty.

The DRAM interface is very similar to the L2 interface. We have an address, a data bus, the usual trio of input enable signals, and an output enable signal.

Stored in DRAM is our actual authoritative protection information, the PKT. Because the protection information is stored in memory, updates to the PKT can be made by the OS by simply updating the table in memory.

The PKT is a two-level hash table where the first table is addressed by (hashed) OID, and the second level by (hashed) PDID. See Figure 6.

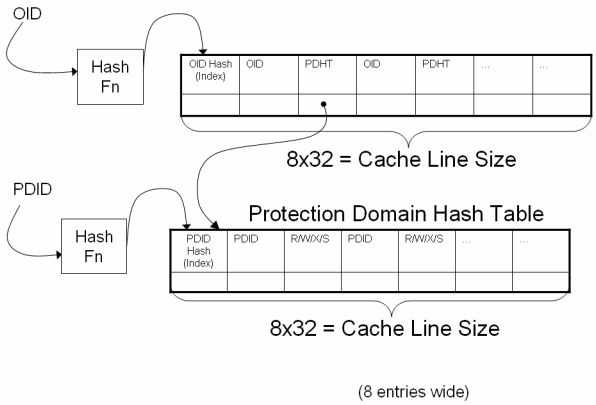

Figure 6: Legba two-level hash table

The base address of the PKT is stored in a new register, the PKTR. This register is used by the PKC to perform any lookups of permissions, but no other caching constructs need to be aware of the PKTR.

The PKT top level is in cache-line sized rows. Since we assume that our OIDs will be sequentially allocated by the operating system, we use a simple 16-bit to 8-bit XOR fold for a hash. This should serve to reduce collisions, particularly if OIDs are allocated in a packed manner; that is, lowest available first.

Each row in the top-level of the PKT has a number of "tries" for hash function collisions. In this initial prototype implementation, we take no steps to avoid hash table collisions beyond our trie limit.

Within each top-level PKT entry, there is a set of OIDs and memory addresses. Each address points to the Protection Domain Hash Table (PDHT) for that OID.

Within the PDHT, there are a set of PDIDs (one for each trie), and a set of 4-bit permissions associated with each one.

The lookup process is actually driven by the PKC on a PKC miss. The lookup process is:

1. Read the cache-line sized row at PKTR+(OID_{high}+OID_{low}).
2. Perform a parallel check of each entry for the correct OID. In cases of multiple matches, use the first. (Since we use 0 for the OID of empty entries, this is possible.)
3. From the associated address A, read the cache-line sized row at A+($PDID_{high}$+$PDID_{low}$).
4. Perform a parallel check of each entry for the correct PDID. Use the associated bits.

One useful trait is that we can make each of the second-level PDHTs a separate object, and give read/write permission over that object to the owner of that object.

E. Changes for Virtual Memory

MiniMIPS does not have virtual memory, and no TLB. Because of this, much of the complexity and design points of Legba are unused. However, most modern processors do have some form of virtual memory.

Legba is best suited to virtually addressed L1 cache. This allows us to remove the TLB from the processor critical path, and to perform OLB lookups in parallel with TLB lookups. Other architectures, while workable, have less ideal choices.

For example, for a physically addressed L1 cache, we need to perform a translation step before a lookup. In this case, we would want our OLB to be at the top level, in the L1 cache. Since our OLB is fully associative and sized to hold all possible 16-bit values, this is a massive addition to the on-core processing.

To implement Legba on a system with virtual memory, certain changes need to be made.

First, the TLB should be real, located in the L2 cache as we have it placed. Second, all memory lookups need to be tagged with an Address Space Identifier (ASID). This is how most modern virtual memory capable processors differentiate different address space data in virtually addressed cache. [1] Thus, we add the ASIDs to the cache.

This introduces the "synonym" problem. We now have differently addressed lines in a single cache that both refer to the same location in memory. Efficient ways of dealing with the synonym problem is an open area of research; most architectures avoid this problem by requiring each address space to be wholly distinct, and any shared memory is bundled into a special "global" ASID.

However, one of the major advantages of Legba is that we can share cache lines between different protection domains – and different address spaces. Using this scheme now requires a different data line for each address-space view of the same data. In this case, Legba is best applied to intra-address space access control, not inter-address space access control.

V. EVALUATION

During our implementation, we encountered several difficulties that were not addressed by the original designers. We also clarified some areas of concern, and can place reasonable constraints on several architectural properties.

A. Timing Concerns

1) Critical Path

Our implementation makes it clear that Legba does not significantly extend the critical path.

First, in most processors, the critical path is established by the EX stage of the pipeline. Legba has no impact on the EX stage.

Secondly, in the case of sidecar hits, the PKC has no impact on either the WB stage or the ID stage. With sidecar hits, the maximum path length of the PKC is a comparator (not equal, of the same width as the OID), 4 AND gates, 2 OR gates, an inverter and a latch. We find it improbable that any critical path could fail to exceed this.

Thirdly, in the case where the critical path is established by the MA stage, our critical path is lengthened by a single AND3 gate.

Finally, for the case where the critical path is established by the IF stage, the critical path is lengthened by only an inverter, a comparator (equals, of instruction width), and an AND gate.

In all cases, we think it is clear that Legba does not significantly impact the processor critical path. Any timing impact from Legba will come from cache misses and pipeline stalls.

2) Frequency of Stalls

Legba contributes to more frequent pipeline stalls in several ways.

First, by adding an OID to the cache lines, Legba reduces the available size of the cache line. Likewise, the PKC takes valuable real-estate on the processor core and reduces the available space for cache. Reduced cache size will always lead to some level of increased miss rate. However, this miss rate

cannot be determined from design, but requires experimental evaluation.

Simply by adding two caches for protection information, Legba adds an additional source for cache misses and pipeline stalls. While we expect PKC misses to be rare compared to data cache misses, this cannot avoid increasing cache miss frequency. Indeed, since we now have 4 caches competing for access to L2, we can have increased stall times from L2 cache contention.

3) Additional Memory Accesses

Aside from increased miss frequency, a cache miss in Legba is potentially more expensive.

On a cache miss from the PKC, we need to index into a two-level hash table. Since this is potentially as much as two DRAM lookups, the stall time is greatly increased, particularly in view of the fact that other processors do not even have a PKC.

In most cases, we think it is probable that L2 will contain the bulk of the top-level portion of the PKT, so we expect most PKC misses to require an L2 lookup followed by a DRAM lookup. We do *not* expect the protection information from the PDHT to ever be found in L2, since the only time it is loaded is when loading the L1 PKC. Thus, unless we flush from the PKC but not from L2, this information will always be found in the PKC, or loaded from DRAM.

Fortunately, we expect PKC misses to be rare. We expect that there will be a limited number of active objects at any time. Since the PKC cache line is approximately 20 bits long, we can expect to hold a large number of object protection lines without substantial cost. Since we would expect normal use to include a small number of tightly clustered object, it should be rare to see many new objects accessed (from different protection domains) in rapid succession. Finally, the addition of sidecars removes the PKC from the critical path. In some cases, it is even possible for the sidecar to hit when the PKC would miss.

B. Unexpected Data Dependency

The original Legba design in [2] contains a flaw. When attempting to write to L1 data cache, we must first lookup the associated OID and send it to the PKC for permissions checking. However, the PKC is in the next pipeline stage!

This problem is inherent in the Legba architectural model. For read operations, this is unimportant. We can read the data, discover that read access is unavailable, and generate an exception. The data is thrown away on exception, so there is no protection impact. For writes, we cannot complete the write until we have validated write permission.

We earlier alluded to a data dependency problem in the data cache and PKC design. In our discussion of their implementation, this point was omitted for clarity.

We developed a workaround for this problem by adding an instruction lookahead from the EX stage to the MA stage. When the MA stage receives a write operation, it uses a 2-state FSM to hold the write until the PKC in the next stage has signaled `wr_ok`. If the next instruction is a memory access, we stall the pipeline until the write permissions are resolved.

One optimization that appears to be available is to perform the write immediately if we discover that the sidecar information is valid and allows writing. Since the sidecar information is validated by comparing the sidecar OID with the cache OID, it seems we can avoid the pipeline stall by checking this in the MA stage.

This is not the case. Our data cache must first *read* the data to obtain the OID. Only after this read can we test the sidecar validity. Unless the processor critical path is more than double the time of a data cache access, we cannot both read the OID and write the data all in a single pipeline stage without noticeably lengthening the critical path.

We consider this problem to have very limited impact on performance. Modern compiler technology is well able to handle re-ordering instructions to avoid known processor hazards. In this case, the only hazard is on a memory write followed immediately by a memory access. Unavoidable cases should be vanishingly rare. In those few cases that do apply, the penalty is only a single cycle stall.

C. Protection Key Table

The PKT as implemented has a limited multi-trie system for handling collisions. While our reasoning is that, due to the order of OID allocations, collisions in the PKT will be limited; we must consider the possibility that our expected usage patterns are incorrect.

Large numbers of collisions would require extending the trie system. However, extending the trie system arbitrarily has the potential to make PKC miss time unbounded. Further, an operating system error has the potential to produce a circular trie list, locking the processor in a hardware memory walker loop. We consider this an unacceptable compromise.

Additional experimental data is necessary before the PKT collision rate can be properly established. Significant additional hardware design would be required to manage extension of the existing trie system.

D. Object Look-aside Buffer

The OLB is effectively a form of very non-standard content addressable memory. In our implementation, we only have 16-bit OIDs; nonetheless, this requires 8 bytes per OID for base and limit, for a total OLB size of 512KB.

Since the OLB is fully associative, searched in parallel on every lookup, this requires two comparators for each line, along with a large aggregation network. We consider the cost of this OLB to be excessive.

The OLB is not a standard CAM, and CAM prices do not apply because of the two-comparator test. However, if we judge from existing CAM architectures, a megabyte of CAM is currently around $10. If we assume that doubling the number of comparators roughly doubles the cost, a rough ball-park figure for this component is $20. With any additional cooling capacity, a realistic total cost could be $30-$50. [12]

While this sounds like a minimal cost, the consumer market has shown a trend toward considering cost above performance (and functionality) when purchasing memory. We consider it unlikely that consumers would be interested in this additional expense without significant demonstrable gain.

VI. Future Work

A. Experimental evaluation of PKT collisions

Our analysis, like that of the original Legba designers, suggests that collisions in the PKT should be rare. We believe that some experimental analysis should be performed to determine the real extent of collisions.

This requires developing a model of a hypothetical operating system using Legba as a basis for protection mechanisms. In the original Legba paper, the authors used user-level code and assigned each variable a different OID. We consider the impact of the operating system to be non-negligible, and we believe that objects must be more sanely delineated.

For example, we could develop a system where a memory allocator controlled its own memory. Allocation requests enter through a call gate; some amount of memory is selected for allocation. That memory unit's PKT contents are reassigned to provide read/write access to the requesting thread.

Developing a design of an entire operating system around the concepts of fine-grained intra-address space protection is a non-trivial task, and a project we consider well worthwhile under its own merits.

B. Per-user Object Look-aside Buffer

One way to fix the problem of a large, fully associative OLB is to have smaller, per-process (or per-user) OLBs. We believe that this can substantially alleviate the problem by reducing the OLB size.

To maintain separation among different processes, this requires flushing the OLB and reloading for each context switch. This is already necessary for virtual memory implementations, where the TLB is flushed and reloaded for each context switch. We anticipate that the OLB flush overhead can be shielded by the TLB flush overhead.

C. Removing Object Look-aside Buffer

Legba's objects are really no different from ordinary segments. They have a base and a limit, and a set of permissions associated with them. The major distinction is that, when accessing segments, the user (via the compiler) must be aware of the segment being accessed.

We believe it is possible to remove the OLB completely by making objects completely into segments. If the program must supply an OID (or segment ID!) on every memory access, this completely removes the OLB from the design. It also changes the semantics of the mapping by allowing objects to overlap. As Legba stands, they may not, since the OLB must be guaranteed to return a single OID for any address.

This is still different from segmentation as applied to modern processors such as the Intel architecture. Segments normally have a set of limited hierarchical privilege levels. We would change this by supporting constructs allowing a completely arbitrary set of access control lists.

This is a significant redesign of the original Legba scheme. We do not believe it is appropriate to simply graft this change onto Legba. On the contrary, we think a change of this nature should necessitate a complete redesign of Legba, because additional optimizations – or problems – may present themselves.

VII. Conclusions

Having implemented Legba, even on a limied architecture such as the MiniMIPS, we believe that this architecture represents a promising direction for research. With some of the limitations we have seen, we do not believe legba is ready for production use in a real microprocessor. However, continuing research is strongly indicated.

In particular, we recommend that two directions toward future work be explored:

First, there is a need to explore operating system design in an arena where fine-grained protection exists.

Second, there should be a redesign of Legba where programs must present OIDs for object access.

References

[1] A. Wiggins, "A Survey on the Interaction Between Caching, Translation and Protection," Tech Report UNSW-CSE-TR-0321, University of South Wales, Sydney, Australia.

[2] A. Wiggins, S. Winwood, H. Tuch, and G. Heiser, "Legba: Fast hardware support for fine-grained protection," 8th Asia-Pacific Computer Systems Architecture Conference (ACSAC).

[3] B. Jacob and T. Mudge, "Uniprocessor virtual memory without TLBs," IEEE Trans. Comput. 50, 5 (May. 2001), 482-499.

[4] E. Koldinger, J. Chase, and S. Eggers, "Architectural Support for Single Address Space Operating Systems," Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, United States, October 1992, pages 175-186.

[5] J. Wilkes and B. Sears, "A comparison of Protection Lookaside Buffers and the PA-RISC protection architecture," Technical Report HPL--92--55, Hewlett-Packard Laboratories, Palo Alto, CA, Mar. 1992.

[6] A. Skousen, D. Miller, "Using a Distributed Single Address Space Operating System to Support Modern Cluster Computing," CSE Dept., ASU, TR-98-007, August 1998.

[7] E. Witchel, J. Cates, K. Asanovic, "Mondrian Memory Protection," Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), October 1--3 2002.

[8] G. Hunt, *et al*, "An Overview of the Singularity Project," Microsoft Research Technical Report MSR-TR-2005-135

[9] H. Bos and B. Samwel, "Safe kernel programming in the OKE," *Proceedings of IEEE OPENARCH 2002.*

[10] B. Bershad, S. Savage, P. Pardyak. "Protection is a Software Issue," Proceedings of the 5th Workshop on Hot Topics in Operating Systems, pages 62—65, 1995

[11] S. Hangouët, S. Jan, L. Mouton, O. Schneider. The miniMIPS project, available online at http://www.opencores.org/projects.cgi/web/minimips/overview

[12] Personal communication, J. Turner, 2005.

[13] C. Small and M. I. Seltzer, "A comparison of OS extension technologies," *USENIX Annual Technical Conference*, pages 41–54, 1996.

[14] George Necula, "Proof-carrying code," Proceedings of the 24th Annual Symposium on Principles of Programming Languages, pp 106-119, 1997

[15] B. Bershad, S. Savage, P. Pardyak, E. Sirer, D. Becker, M. Fiuczynski, C. Chambers, S. Eggers. "Extensibility, Safety and Performance in the SPIN Operating System," *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, pp. 267--284.

[16] M. Swift, B. Bershad, H. Levy. "Improving the Reliability of Commodity Operating Systems," *SOSP 2003*.

[17] T. Chiueh, G. Venkitachalam, P. Pradhan. "Integrating segmentation and paging protection for safe, efficient and transparent software extensions," Proc. ACM SOSP 1999

[18] G. Graham and P. Denning. "Protection – Principles and Practice," In AFIPS Conference Proceedings Volume 40 Spring Joint Computer Conference, Montvale, New Jersey, 1972

[19] Intel Corp. "Itanium Architecture Software Developer's Manual," Feb 2000. URL http://developer.intel.com/design/itanium/family.

[20] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. "An empirical study of operating system errors," *Proceedings of the 18th ACM Symposium on Operating Systems Principles, pages 73–88, Oct. 2001*.