Legba: Fast Hardware Support for Fine-Grained Protection

Adam Wiggins[1], Simon Winwood[1], Harvey Tuch[1], and Gernot Heiser[1,2]

[1] University of New South Wales, Sydney 2052, Australia
[2] National ICT Australia, Sydney, Australia
{awiggins,sjw,htuch,gernot}@cse.unsw.edu.au

Abstract. Fine-grained hardware protection, if it can be done without slowing down the processor, could deliver significant benefits to software, enabling the implementation of strongly encapsulated light-weight objects. In this paper we introduce Legba, a new caching architecture that aims at supporting fine-grained memory protection and protected procedure calls without slowing down the processor's clock speed.
This is achieved by separating translation from protection, which allows the use of virtually-addressed caches and moving the TLB off-core. Protection is implemented in two stages. We add protection information in the form of an object ID to each cache line. This object ID is combined with a per-protection context identifier, and the result is used to index into a protection cache, which delivers the access rights. As no range check is required on the protection cache, it can be set-associative, allowing it to be made large, fast and low-power, compared to a fully associative TLB. On a cache miss, the object ID is retrieved in parallel to the cache line fetch, performing the protection range check off-core.
A new switch permission enables Legba to implement protected procedure calls, where the new context identifier is taken from the instruction cache line's object ID. This mechanism is similar to call gates but more flexible. The paper compares Legba with approaches based on the idea of a protection look-aside buffer, in particular with respect to coverage.

1 Introduction

Mobile code is becoming increasingly widespread, and thus secure execution of untrusted code is presents a significant challenge to modern computer systems [1]. As well, dynamic extensibility has long been promoted as a way to manage the complexity, and improve maintainability and reliability of operating systems [2–5]. Recently, the low reliability of some system components, particularly device drivers, has triggered renewed efforts to isolate such components [6, 7].

The common problem here is the need to isolate untrusted (buggy or potentially malicious) code. In addition, component technology [8–10], which is an attractive way of constructing extensions, is leading to a reduced granularity of the units of code and data that require protection or isolation [11].

While a memory-management unit (MMU) provides mechanisms for implementing protection and isolation, attempts to use these for mobile code or OS extensions has in the past generally lead to poor performance [1], mostly resulting from the high cost of protection domain crossings (i.e. context switches). This has lead to a widespread employment of pure software techniques for protection and isolation of extensions [3, 12–15]. These approaches are generally justified with the high cost and coarse granularity of hardware-based protection.

This high cost is not unavoidable. Even on present hardware, careful design and implementation of OS primitives can reduce the cross-domain invocation cost to within a single order of magnitude of that of a normal function call [16, 17]. While this still constitutes significant overhead on primitive operations, in terms of overall system execution times this is often reduced to a few percent [18, 19]. Still, the overheads may be too high for component software with high invocation frequencies.

However, software-only protection has its cost too: run-time checks cannot be avoided unless restrictive programming models are imposed, and the size of the *trusted computing base* (TCB) dramatically increases due to the inclusion of compilers and language runtime systems. Perhaps most critically, a single security flaw in a system employing software-only protection will generally provide an attacker with the full privileges of the underlying virtual machine [11].

Hardware mechanisms would be the preferred means of providing protection or isolation, if they provided finer granularity and if the cost of context switches could be reduced compared to present processors.

This paper presents *Legba*, a new protection cache architecture, which is designed to reduce the granularity of protection, without limiting the processor's clock rate. Legba furthermore supports a *protected procedure call* [20,21] mechanism which allows a program to change its protection domain in a controlled manner without the need to enter the operating system (OS) kernel. This enables fast protected component invocation.

The reminder of this paper is organised as follows. Section 2 presents related work, Section 3 introduces our proposed Legba architecture. Section 4 describes the experimental setup we used, and Section 5 presents the results, followed by conclusions and future work in Sections 6 & 7.

2 Related Work

2.1 Translation Look-aside Buffer

Current processors employ a *translation look-aside buffer* (TLB), which caches page translations as well as access rights. In order to allow sharing of the TLB between different processes, and thus reduce context switching costs, the TLB is usually tagged with an *address-space identifier* (ASID). The ASID of the currently active process is stored in a processor register and concatenated with the virtual address on a TLB lookup.

Making protection more fine-grained in such a system would mean reducing the page size. Small page sizes, however, imply more memory-management overhead in the OS, and reduced I/O performance when paging. The trend in modern operating systems is towards *larger* rather than smaller page sizes. As a single page size is anyway unlikely to provide good performance under all circumstances, TLBs of modern architectures support a range of page sizes. Multiple page sizes, however, are in general implemented via a fully-associative TLB [22]. Since large fully-associative caches are slow and energy hungry, and since the TLB is on the processor core, TLB capacity is generally limited to, at most, a few hundred entries. Consequently, TLB coverage is inherently limited, and would be further degraded by smaller page sizes.

The inadequate coverage of modern TLBs has been highlighted by several studies [23–26]. Several attempts have been made to address this, including *super-pages* [22], *sub-blocking* [27], *in-memory translation* [28], *virtually-addressed memory hierarchies* [29,30], *in-cache translation* [31], and even *software-managed address translation* [32]. However, all these studies focused on improving translation coverage, while protection issues have at best been a secondary consideration.

2.2 De-coupling Protection From Translation

Given the conflicting requirements on the granularity of translation (which should be large in order to maximise translation coverage) and protection (which should be small), it makes sense to consider separating the hardware mechanisms for protection and translation.

One such approach is that used in the PA-RISC [33] and Itanium [34] processors. These tag TLB entries with a *protection-key*, which is used to look up additional access information in a separate protection cache. On the Itanium this cache is a small (16 on the first generation processor) fully-associative set of *protection-key registers* (PKRs) without context-specific tags.

The small size of the PKR file is probably a result of the lookup being on the critical path and the lack of a context tag, which means that the PKRs must be invalidated or reloaded on a context switch. However, there is no obvious inherent limitation on the size of the PKR file, as it could be made set-associative and tagged with a context ID.

The main advantage of protection keys is that they allow sharing TLB entries of shared pages, even if different context have different rights to the page, thus somewhat increasing TLB coverage in the presence of sharing [35]. However, protection keys do not support protection at sub-page granularity and only partially decouple protection from translation. Furthermore, they require an additional cache (the PKRs) on the processor core (although the lookup latency can be hidden in the pipeline) and the TLB remains on the processor core.

An alternative approach, the *protection look-aside buffer* (PLB), completely decouples protection and translation [36]. In this scheme, all protection data is removed from the TLB, which can then be moved off-core if a virtually-addressed L1 cache is used. The PLB is essentially a TLB with no translation information (making it smaller), and thus has essentially the same drawbacks as a classical TLB: it is in the processor's critical path, and the need to support a range of protection granularities implies that it is fully associative. Hence, its speed and capacity (and thus coverage) are limited in the same way as a TLB.

The recently proposed *Mondrian memory protection* (MMP) [37] addresses some of these shortcomings. It assumes a single, shared (virtual or physical) address space with access rights defined by per-context *permissions tables*. A PLB is used to cache these rights. In order to move the PLB off the critical path, Witchel et al. introduce the concept of *sidecar registers*, which are associated with each of the processor's registers able to hold addresses. These sidecars cache the base, limit and access rights of the last memory reference via those registers, utilising locality of pointer references. The sidecars reduce the frequency of PLB accesses and have the advantage that the segment information they hold does not need to be aligned to any particular block size. Unlike PLB entries, the sidecars are not tagged with a protection-domain ID, and thus need to be flushed on a context switch.

2.3 Protected Procedure Calls

The idea of protected procedure entry points goes back to Multics *call gates* [20], which were a transparent, secure mechanism for increasing a process's privileges. Similar mechanisms exist on the x86 [38] and Itanium [34] architectures. These are tied to the hierarchical privilege model supported by these architectures. The hierarchical model has proven to be inflexible, and, with one recent exception [39], no operating system on x86 uses more than two privilege levels.

The IBM System/38 generalised call gates into a mechanism, called *profile adaptation* [40], for executing encapsulated (but not necessarily privileged) code. This mechanism is highly dependent on System/38's capability-based protection model. Recently, a protection domain *switch* mechanism was proposed for the Sombrero single-address-space architecture [41]. The design uses a PLB generalisation, called the *range protection look-aside buffer* (RPLB), in order to cache access rights, including for protection-domain switches. The Sombrero design requires an RPLB entry for each caller-entry-point combination, which uses up the RPLB very quickly. Furthermore, the RPLB is unlikely to scale to high clock speeds.

3 The Legba Architecture

3.1 Principle of Operation

What really limits performance, and thus the ability to apply protection at a fine granularity, in schemes designed around some form of TLB or PLB is the need to perform an associative lookup of an address *without knowing the base address* of the object whose attributes are cached. Essentially, a TLB or PLB is limited by the need to mask the page size in order to obtain the base address.

Any effective solution must avoid a cache lookup for an unknown base address (a range check). This can be achieved by *associating the protection information with each I/D-cache line*.

Placing the actual protection bits in each cache line has been proposed before [42]. However this approach makes the protection bits global (i.e., independent of the protection context) which can only be avoided by either flushing the caches on a context switch, or adding a PDID tag to them. In addition, protection updates require that each cache line's permissions be updated in a sequential manner.

The main idea behind Legba is to eliminate the range-check problem by adding a level of indirection. While this has the potential to increase costs, we will show that it will, in fact, make fine-grained protection feasible, by trading transistors for clock speed.

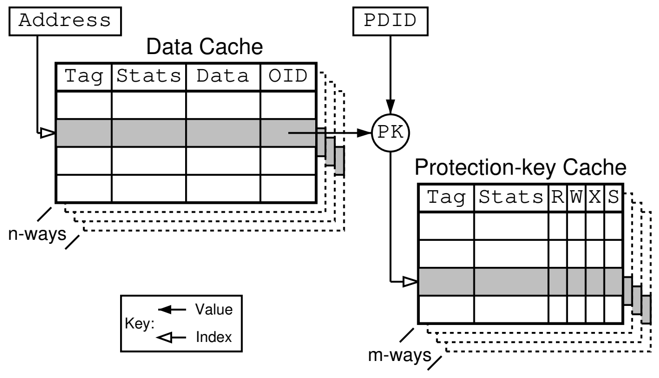


Fig. 1. Legba cache architecture.

Fig. 1 shows the main features of the architecture. We tag cache lines with an *object identifier* (OID). On a cache hit, the OID is concatenated with the *protection-domain identifier* (PDID) of the presently executing process. The result is used to look up a *protection-key cache* (PKC) which holds the protection bits. Neither cache lookup requires a range check, and there is no need for the PKC to be fully associative, allowing it to be large with less limitations on its speed.

3.2 Protection-Key Caching

The PKC index should be generated from the OID only, in order to support inexpensive flushing of object accesses. This creates a potential for high collision rates where objects are heavily shared between many protection domains, suggesting that the PKC's associativity should be reasonably high.

As the object name space is completely separate from the address space, it is possible to re-tag objects dynamically (i.e. change their OID). Generating the PKC indexing solely from the OID supports dynamic re-tagging, which can then be used by the operating system to "re-colour" objects if a high rate of PKC collisions is detected.

On a miss, the PKC must be reloaded from a *protection-key table* (PKT). This is organised as a two-level hash table, where the OID is used to index the first table, the *object hash table*. This contains a pointer to the second table, the *protection-domain hash table*, which is indexed by the PDID. This lookup can be done by a fast hardware walker; on a miss in either table, a software exception is raised.

These tables are themselves memory objects, and can be cached like any data, similar to hardware-walked page tables on some architectures. Since they are memory objects, these hash tables themselves are protected by Legba memory protection. Among others, this means that object "ownership" can be given to user code by giving it write access to the protection-domain hash table of the object. The owner can then update the access control list of the object by modifying its protection-domain hash table.

3.3 Instruction and Data Cache Misses

Storing the object ID in the cache line slightly complicates cache miss handling. The hardware not only must fetch the cache line, but also the OID. However, since the cache data and the OID lookup utilise the same address, they can be done in parallel, potentially allowing the OID lookup latency to be hidden by the cache line fetch.

This relaxation of time constraints allows the use of a large fully-associative cache to implement ($address \rightarrow OID$) mappings. This cache is called the *object look-aside buffer* (OLB). It can support multiple page sizes, or even a more expensive (*base*, *limit*) form of segmentation.

The design space contains further alternatives. For example, the OID mapping could be held in lower-level caches, with software miss handling similar to *software-managed address translation* [32].

3.4 Protected Procedure Calls

In addition to the familiar (R)ead, (W)rite and e(X)ecute rights, Legba also supports a (S)witch permission, similar to that proposed for Sombrero [41], which guards protected procedure call objects.

A protection-domain crossing in Legba requires two interlocked instructions, which can be viewed as a replacement for the syscall or trap instructions supplied by most architectures. The first is a branch-linked-locked instruction, which differs from a normal branch-linked by additionally setting a condition flag. That flag requires the branch target to contain a switch-load instruction, otherwise an exception is raised. Execution of a branch-linked-locked instruction, like any other instruction, requires X permission to the code object containing the instruction.

The switch-load instruction marks an entry point into a protection-domain. Unlike other instructions, its execution requires the S permission to the code object it resides in. The instruction performs a normal load of a general-purpose CPU register (GR), but at the same time loads the OID of the code object into the PDID register. Thus, the execution of the branch-linked-locked/switch-load pair changes the protection context of the executing thread. The GR load can be used to set up the stack pointer for the execution of the protected procedure.

3.5 Pipeline Implementation

Since a protection-key cache lookup depends on the OID (obtained from the instruction or data cache) it should be located in the pipeline after the respective cache. Fig. 2 shows an example Legba pipeline. Here, a single-issue 5-stage in-order pipeline is employed with split instruction and data caches (ICache, DCache) as well as split instruction and data protection-key caches (IPKC, DPKC). The IPKC handles the execution-type access rights of X and S, while the DPKC handles the data-access rights of R and W. The IPKC also controls the loading of the current PDID register via the switch operation.

Because the protection-key caches reside in the pipeline stages after the cache references, access permission faults will incur a single cycle delay penalty in addition to the exception handling overheads.

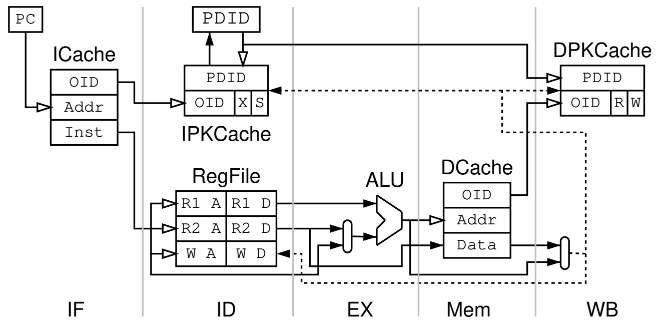


Fig. 2. Example Legba pipeline.

For out-of-order pipelines, protection cache (either PLB or PKC) lookups do not have to complete until instruction retirement, effectively removing them from the processor's critical path. However, the lower access time of the PKC should enable higher instruction retirement rates, compared to a PLB-based design.

3.6 Sidecar Implementation

To reduce the number of PKC accesses, Legba supports a sidecar optimisation similar to that proposed in Witchel et. al. [37]. During a memory reference, the OID stored in the sidecar is compared to the one stored in the cache line. On a match the permission bits from the sidecar are used, avoiding a PKC access.

As opposed to the range check required by MMP, Legba's sidecars require a simple comparison of the OID. This implies that Legba's register file with sidecars is at most half of the size the Mondrian model. This size difference should lead to a lower access energy, which is significant considering the sidecars are accessed on every instruction, potentially multiple times.

3.7 Summary

Legba can be regarded as a two-stage PLB. The first stage associates an address range with an object. Because this stage is off-core and only invoked on a data cache miss, the cost of its range-check can be shielded by the expense of the cache miss[3]. The second stage associates the object and current protection-domain context with an access permission. This stage is on-core and accessed on every cache reference, to validate the access rights of the reference. The key consequence of the architecture is that the range check is removed from the core, facilitating the use of protection caches with increased coverage at lower energy consumption and higher access speeds.

[3] The cost of the OLB range-check is only shielded by a cache miss if the access hits in the OLB.

Legba works equally well for virtually-addressed as for physically-addressed caches. However, it is most powerful when used with virtually-addressed caches, as this completely removes the TLB (and thus any range check) from the processor's core. This is unlike traditional architectures, where (even with virtual caches) a TLB lookup is still required to obtain the protection information.

The cost of Legba is an increase in first-level I/D-cache size to accommodate OIDs, a similar increase in bandwidth requirements for cache line fetches, and the addition of an off-core OLB. The benefits are increased coverage of the protection-key cache over TLB- or PLB-based approaches.

4 Experimental Evaluation

The performance evaluation of Legba was done in three stages. Firstly, we generated memory reference traces for the MediaBench [43] benchmark suite running on a simulated ARM [44] processor.

One of the motivations for this work was fine-grained protection on high-performance embedded systems, where processors such as the ARM are common.

Secondly, these traces were fed into a cache-level simulator for a number of cache architectures: a hypothetical ARM system for the baseline, a Legba system, and a PLB-based system. Each configuration was run for a range of protection granularities and protection-cache sizes.

Finally, a cache modeller was used to generate timing and energy profiles for each architecture and configuration.

The remainder of this section is organised as follows: Section 4.1 introduces the simulation environment in more detail, while Section 4.2 describes the different cache architectures. Section 4.3 outlines the benchmarks used and their protection characteristics, and Section 4.4 discusses the anticipated differences between the simulation environment and a real implementation.

4.1 Simulation Environment

We used the SimpleScalar [45] ISA simulator to generate a set of memory traces. SimpleScalar simulates the user-level portion of a system, forwarding all system calls to an OS emulation layer inside the simulator, emulating Linux in this case. This results in traces which do not include any OS interference, especially cache pollution.

To simulate the various cache architectures, we separated SimpleScalar's cache functionality into a separate cache simulator called *tracesim*. This simulator takes in a memory trace, a set of cache parameters, and a set of object descriptors, and generates cache statistics for each combination of cache parameters. The simulation output (number of hits and misses for each cache) was then processed by the CACTI [46] cache modeller and combined with the time and energy characteristics of SDRAM to produce a total energy consumption and runtime. The time and energy characteristics of the register file and sidecar registers, as well other parts of the processor, were not modelled.

The granularity of objects and protection domains was varied to examine the behaviour of these systems under different protection scenarios. Accordingly, PDIDs and OIDs were generated as follows: for the finest grain of protection domains, each function was assigned a separate PDID. For the coarse grain protection domains, the application code was assigned one PDID, while any libraries — primarily libc — were assigned another. Finally, to provide fine-grained OIDs each program variable, whether dynamic or static, was assigned a unique OID.

4.2 Simulation Configuration

Each cache configuration was based on a hypothetical ARM processor modelled after Intel's XScale processor, with characteristics as in Table 1.

Table 1. Baseline configuration

Parameter	Value
Clock speed	600MHz
I-TLB	32-entry, fully assoc.
D-TLB	32-entry, fully assoc.
I-Cache	32k, 32-way, 32byte line size
D-Cache	32k, 32-way, 32byte line size
Pagesize	4kB
TLB-reload	hardware; 2-level page table
Memory	100MHz SDRAM

To simplify the simulation, system data structures, such as page tables, were simulated as being loaded directly from memory. We also assume that a cache write-back will require another translation to obtain the physical address.

In the Legba and PLB configurations, a protection table lookup was assumed to use the minimum number of memory accesses required by the destination object's size.

For the baseline configuration, the TLB is accessed in parallel to the cache in order to check permissions.

The Legba configuration is shown in Table 2. As Legba provides an alternate protection mechanism, the TLB is not required on a cache access, and so was moved off-core. Sidecar registers were consulted on every memory access (whenever an instruction was fetched, the PC's sidecar was consulted), and the respective protection key cache was only accessed on a sidecar miss. On a cache miss, the OLB was consulted. To model protection granularity down to that of a single word, an OID was stored per word in the cache line, as a result the OLB could be consulted multiple times per cache miss if the OLB entry did not map the entire cache line. This form of word-level granularity is not particularly compact, leaving room for improvement.

In order to model a protection domain-switch, the I-PKC was accessed and the sidecars were flushed each time the current protection domain changed (via executing an instruction tagged with a different PDID).

Table 2. Legba and PLB configurations

Architecture	Parameter	Value
Legba	I-PKC	128-, 256-, 512-, and 1024-entry, 32-way
	D-PKC	128-, 256-, 512-, and 1024-entry, 32-way
	OLB	128-entry, fully-assoc.
PLB	I-PLB	32-, 64-, and 128-entry, fully-assoc.
	D-PLB	32-, 64-, and 128-entry, fully-assoc.
	S-PLB	32-, 64-, and 128-entry, fully-assoc.

The PLB configuration is also shown in Table 2. As with Legba, the TLB was also moved off-core. Sidecar registers were consulted on every memory reference, with a miss going to the respective PLB. If the result was a PLB hit, the PLB's super-page entry was copied into the sidecar. On a PLB miss, the segment's base and limit were copied into the sidecar and the largest power of two page of the access region loaded into the PLB.

To provide an equivalent to Legba's switch instruction for the PLB case, a Switch-PLB (S-PLB) was simulated. This caches the destination PDID and permissions, and is tagged with the entrypoint and current PDID. The S-PLB is accessed whenever the PDID changes. While the S-PLB can be set associative, preliminary benchmarks suggested that the only practical implementation would be fully associative, as lower associativities resulted in significantly reduced performance due to *conflict misses*.

On an S-PLB miss, a 3-level protection table lookup was assumed. As with Legba, the sidecars were flushed on a PDID change.

4.3 Benchmarks

The MediaBench [43] suite was chosen as representative of embedded applications with a relatively short run time — a desirable characteristic given the simulation overhead. Table 3 shows the number of protection domains, object IDs, and protection domain switches for each benchmark. The benchmarks presented in Section 5 were chosen as those exhibiting interesting, representative, or significant behaviour.

4.4 Simulation Accuracy

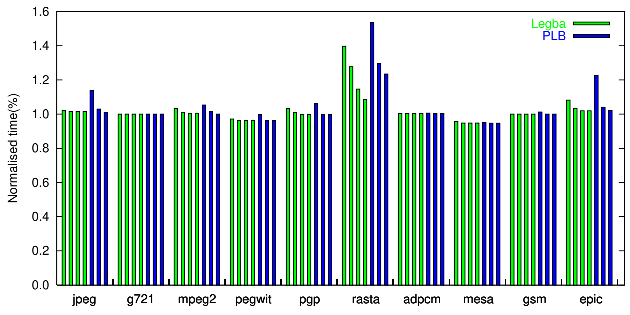
Although our simulation attempts to mirror a real system, time and complexity constraints meant that some aspects of the system had to be simplified.

The primary simplification was that all tables would be loaded from main memory. We expect a real system to reload the protection caches (the PKCs and PLBs) from in-cache tables, with the TLB and OLB being loaded from main memory.

We believe, however, that our results are still significant; the trends discussed in Section 5 are inherent characteristics of the two models, and the loading of protection entries from memory will not significantly effect our results.

Table 3. Properties of benchmarks

	No. Protection Domains & Context Switches				
Benchmark	Coarse	context switches	Fine	context switches	OIDs
jpeg	2	13070	866	350681	449
g721	2	6922258	639	25457220	442
mpeg2	2	5813793	746	23286653	652
pegwit	2	1835251	719	4507475	476
pgp	2	1122637	1087	11526264	782
rasta	2	6663146	1000	15959160	1366
adpcm	2	665126	626	679288	433
mesa	2	12885420	1635	37622046	604
gsm	2	281158	732	17631406	489
epic	2	194591	657	965314	575



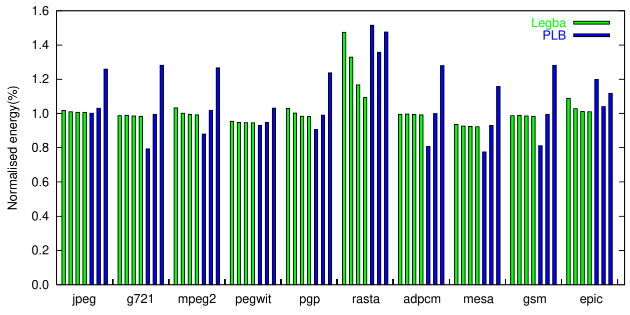


Fig. 3. Coarse-grained PDIDs and fine-grained OIDs (left to right: 128-, 256-, 512-, 1024-entry PKCs, 32-, 64-, 128-entry PLBs) without sidecars (top: execution time, bottom: energy).

5 Results

The results for run time and run energy are presented, normalised to the baseline configuration. Fig. 3 shows the time and energy performance for coarse-grained PDIDs and fine-grained OIDs, when no sidecar registers are employed. While Legba's time and energy results improve with increased PKC size, the PLB exhibits a tradeoff between time and energy. In most cases the 32-entry PLBs have insufficient coverage, requiring at least 64-entry and sometimes even 128-entry PLBs to match Legba's performance, The energy results show the inverse with the larger PLBs using more power; the 128-entry PLB being generally about 25% more power-hungry than a Legba configuration.

In nearly all cases the 32-entry PLB uses less energy than all the Legba configurations. This is due to the energy overheads of reading out the OIDs from the I/D caches on each cache reference[4]. The 64-entry PLBs then lose this advantage, levelling out with the Legba configurations while the 128-entry PLBs use more energy in every case.

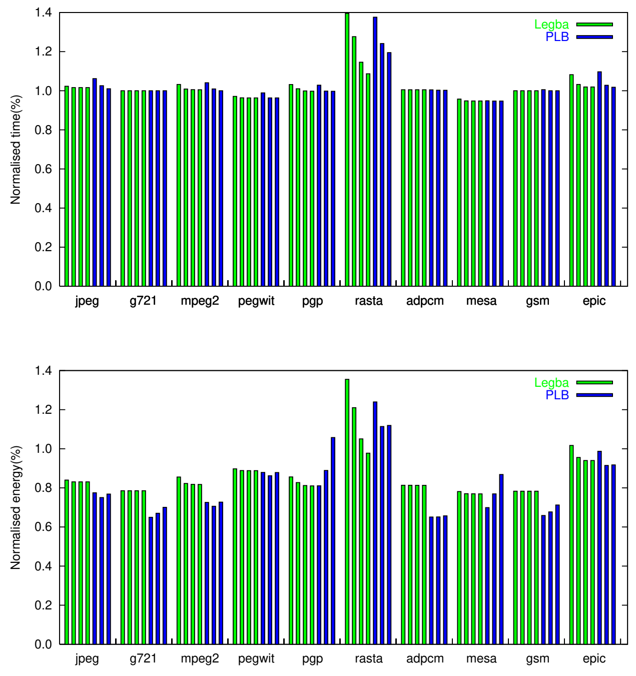


Fig. 4. Coarse-grained PDIDs and fine-grained OIDs, with sidecars.

Fig. 4 shows the same results with sidecars added. While on average performance increases only marginally, energy shows a significant decrease. The shielding of the protection caches (PLB or PKC) by the sidecars causes the core's energy to be dominated by the cost of I/D cache references for most benchmarks. Exceptions are benchmarks, like Rasta, where a larger number of objects are referenced, and a large number of protection table lookups are pushing up the energy costs.

[4] We believe the energy overheads of reading the OIDs from the caches were overstated due to limitation of simulating them in CACTI.

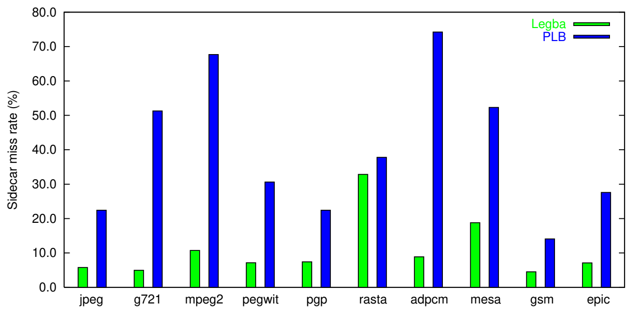


Fig. 5. Sidecar miss rates for largest PKC and PLB, coarse-grained PDIDs and fine-grained OIDs.

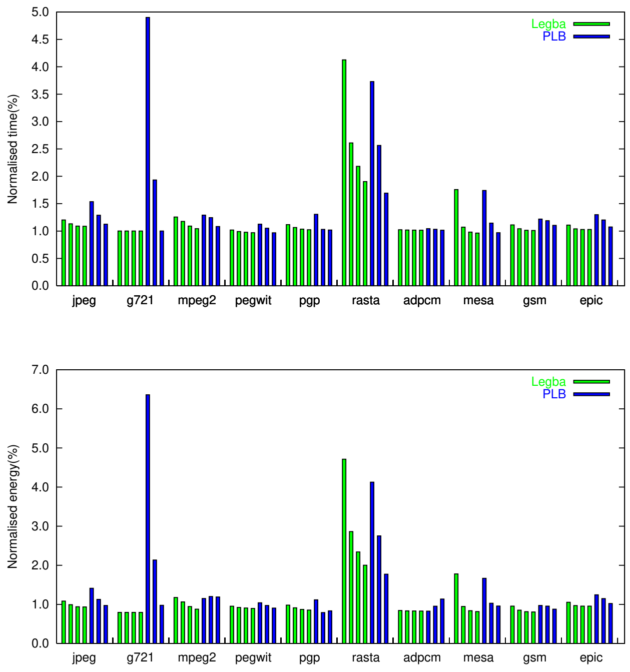


Fig. 6. Fine-grained PDIDs and OIDs, with sidecars.

As expected, Legba shows consistent improvements in energy consumption with increasing PKC sizes, as the energy consumption of the PKC itself is quite insignificant. The PLB results, while generally showing a decrease in energy consumption with increasing PLB size, show several cases where the opposite is true. This shows that the optimal PLB size, with respect to energy, is quite application dependent.

Fig. 5 shows the sidecar miss rates for the largest PKC/PLB configurations. Legba's sidecars clearly exhibit much higher hit rates. This is a result of how sidecars are loaded on a miss: as PLB entries are aligned power-of-two ranges, in many cases several PLB entries are required for a single segment, leading to sidecars not covering the complete segment after a reload form the PLB. Only on a PLB miss are full (*base*, *limit*) entries loaded into the sidecar.

Fig. 6 shows the results for both fine-grained objects and protection-domains. On average Legba's increased coverage manages fine-grained protection more effectively. Out of the PLB configurations, only the 128-entry one consistently approaches Legba's performance, however it nearly always fares worse in terms of energy. The exception is the Rasta benchmark, where Legba's increased table accesses cause it to be both slower and use more energy.

Fig. 7 shows, once again, that the Legba's sidecar miss rate is much lower than the PLB setup.

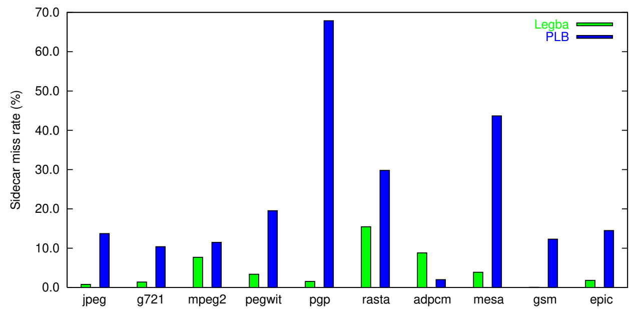


Fig. 7. Sidecar miss rates for largest PKC and PLB, fine-grained PDIDs and OIDs.

To try and get a feel for the how 64-bit addressing would impact performance we reran the time and energy simulations using a 5-level protection table to reload both Legba's OLB and the PLBs. The results showed little variation to the 3-level table. Besides fairly consistent degradation in time and energy for both Legba and the PLB, the only notable result was that for fine-grained PDIDs and OIDs, Legba's time and energy for the Rasta benchmark out-performed that of the PLB.

6 Conclusions

In this study we have introduced the Legba cache architecture for fine-grained protection and evaluated its time and energy performance to that of the PLB. The results show that Legba's protection caches scale more effectively than the PLB organisation. In particular while increases in the size (and hence coverage) of the PKC show modest increases in energy and time costs. Similar increases in PLB coverage need to be weighted against the significant energy and time impact of their fully-associative nature.

One of the most significant result of the study has been to show that with the use of MMP's sidecar registers, Legba or PLB based protection combined with an off-core

TLB makes fine-grained protection cheaper in both energy and time (for the majority of the benchmarks evaluated) than a on-core TLB with only page-based protection.

We also show that Legba's sidecars are simpler and have lower miss rates than MMP's range-based sidecars. However, one drawback of the Legba approach that has limited its performance in this evaluation environment has been the cost of additional memory accesses over the PLB. Because both the OLB and PLB were loaded from hardware walked tables, the overhead of a PLB miss is negligible compared to that of a PKC miss, as both require on average two memory references. A major focus of future work will be to reduce this overhead through more intellegent OID mapping tables and protection-key tables.

7 Future Work

While the results of this evaluation have shown Legba to an attractive architecture for fine-grained protection environments, a number of limitations in the both the evaluation and architecture still need to be addressed.

A major limitation of the evaluation was the lack of any OS modelling, and that the protection and translation tables were loaded from main memory and not the caches. This leaves significant uncertainty about the overheads that both the Legba and PLB architectures will incur in a real system. In addition, any future studies will need to look at the effects of 64-bit architectures and software loaded tables.

Further work is required to investigate the ability of the Legba architecture to provide effective support for word- and even byte-grained protection. We are currently exploring a number of approaches resulting in small, constant increase in the size of the L1 caches. A related issue is that of OLB organisation. The segmented (*base*, *limit*) nature of protection attributes suggest that OLBs with some form of support for segmentation in the OLB would improve its hit rate, particularly if multiple segments could be mapped per OLB entry. Again we are currently exploring a number of designs that provide just this.

References

1. Drew Dean and Edward W. Felten. Secure mobile code: Where do we go from here? In *DARPA Workshop on Foundations for Secure Mobile Code*, Monterey, CA, USA, Mar 1997.
2. W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Comm. ACM*, 17:337–345, 1974.
3. Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. 15th ACM SOSP*, pages 267–284, Copper Mountain, CO, USA, Dec 1995.
4. M.I. Seltzer, Y. Endo, C. Small, and K.A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. 2nd USENIX OSDI*, pages 213–228, Nov 1996.
5. Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. 15th ACM SOSP*, pages 251–266, Copper Mountain, CO, USA, Dec 1995.

6. George Candea and Armando Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proc. 8th HotOS*, pages 125–130, 2001.
7. Michael M. Swift, Steven Marting, Henry M. Levy, and Susan G. Eggers. Nooks: An architecture for reliable device drivers. In *Proc. 10th SIGOPS European WS.*, pages 101–107, St Emilion, France, Sep 2002.
8. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley/ACM Press, Essex, England, 1997.
9. The component object model specification. Technical report, Microsoft Corporation and Digital Equipment Corporation, 1995. http://www.microsoft.com.
10. Corba components. TC Document orbos/99-02-05, Object Management Group, Mar 1999. ftp://ftp.omg.org/pub/docs/orbos/99-02-05.pdf.
11. Trent Jaeger, Jochen Liedtke, and Nayeem Islam. Operating system protection for fine-grained programs. In *Proc. 7th USENIX Security Symp.*, pages 143–157, San Antonio, Tx, USA, Jan 1998.
12. Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in Java. In *Proc. 1998 USENIX Techn. Conf.*, pages 259–270, New Orleans, USA, Jun 1998.
13. Michael Golm, Jürgen Kleinöder, and Frank Bellosa. Beyond address spaces: Flexibility, performance, protection, and resource management in the type-safe JX operating system. In *Proc. 8th HotOS*, pages 3–8, Schloß Elmau, Germany, May 2001.
14. Brian N. Bershad, Stefan Savage, Przemyslaw Pardak, David Becker, Marc Fiuczynski, and Emin Gün Sirer. Protection is a software issue. In *Proc. 5th HotOS*, Orkas Island, WA, USA, May 1995.
15. Laurent Daynès and Grzegorz Czajkowski. Lightweight flexible isolation for language-based extensible systems. In *Proc. 28nd VLDB Conf.*, Hong Kong, China, 2002.
16. Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Herrman Härtig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. Achieved IPC performance (still the foundation for extensibility). In *Proc. 6th HotOS*, pages 28–31, Cape Cod, MA, USA, May 1997.
17. Takahiro Shinagawa, Kenji Kono, and Takashi Masuda. Exploiting segmentation mechanism for protecting against malicious mobile code. Technical Report 00-02, Dept. of Information Science, University of Tokyo, May 2000.
18. Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proc. 16th ACM SOSP*, pages 66–77, St. Malo, France, Oct 1997.
19. Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. In *Proc. 5th USENIX OSDI*, Boston, MA, USA, Dec 2002.
20. Jerome H. Saltzer. Protection and the control of information sharing in Multics. *Comm. ACM*, 17:388–402, 1974.
21. Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. In *Proc. 12th ACM SOSP*, pages 102–113, Dec 1989.
22. Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. Tradeoffs in supporting two page sizes. In *Proc. 19th ISCA*. ACM, 1992.
23. J. Bradley Chen, Anita Borg, and Norman P. Jouppi. A simulation based study of TLB performance. In *Proc. 19th ISCA*. ACM, 1992.
24. Jerry Huck and Jim Hays. Architectural support for translation table management in large address space machines. In *Proc. 20th ISCA*, pages 39–50. ACM, 1993.
25. Madhusudhan Talluri. *Use of Superpages and Subblocking in the Address Translation Hierarchy*. Phd thesis, University of Wisconsin-Madison Computer Sciences, 1995. Technical Report #1277.
26. Gokul B. Kandiraju and Anand Sivasubramaniam. Characterizing the d-TLB behavior of SPEC CPU2000 benchmarks. In *Proc. ACM SIGMETRICS*, 2002.

27. Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB performance of superpages with less operating system support. In *Proc. 6th ASPLOS*, pages 171–182, San Jose, CA, USA 1994.
28. Patricia J. Teller. Translation-lookaside buffer consistency. *Trans. Computers*, 23:26–36, 1990.
29. Xiaogang Qiu and Michel Dubois. Options for dynamic address translation in COMAs. In *Proc. 25th ISCA*, pages 214–225, 1998.
30. Xiaogang Qiu and Michel Dubois. Towards virtually-addressed memory hierarchies. In *HPCA*, pages 51–62, Jan 2001.
31. David A. Wood, Susan J. Eggers, Garth Gibson, Mark D. Hill, Joan M. Pendleton, Scott A. Ritchie, George S. Taylor, Randy H. Katz, and David A. Patterson. An in-cache address translation mechanism. In *Proc. 13th ISCA*, pages 358–365, 1986.
32. Bruce Jacob and Trevor Mudge. Uniprocessor virtual memory without TLBs. *Trans. Computers*, 50:482–499, 2001.
33. Ruby B. Lee. Precision architecture. *IEEE Comp.*, 22(1):78–91, Jan 1989.
34. Intel Corp. *Itanium Architecture Software Developer's Manual*, Feb 2000. URL http://developer.intel.com/design/itanium/family.
35. Matthew Chapman, Ian Wienand, and Gernot Heiser. Itanium page tables and TLB. Technical Report UNSW-CSE-TR-0307, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, May 2003.
36. Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. Architectural support for single-address-space operating systems. In *Proc. 5th ASPLOS*, pages 175–86, 1992.
37. Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proc. 10th ASPLOS*, Oct 2002.
38. Intel Corp. *IA-32 Architecture Software Developer's Manual*, 2002. URL http://developer.intel.com/design/pentium4/manuals.
39. Tzi-cher Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proc. 17th ACM SOSP*, pages 140–153, Kiawah Island, SC, USA, Dec 1999.
40. Viktors Berstis. Security and protection in the IBM System/38. In *Proc. 7th Symp. Comp. Arch.*, pages 245–250. ACM/IEEE, May 1980.
41. Alan C. Skousen and Donald Miller. Resource access and protection in the Sombrero protection model, software protection data structures and hardware range protection lookaside buffer. Technical Report TR-95-013, Computer Science and Engineering Department, Arizona State University, May 1996.
42. Bruce Jacob and Trevor Mudge. Software-managed address translation. In *Proc. 3rd HPCA*, pages 156–167, 1997.
43. Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and commu nicatons systems. In *Proceedings of the thirtieth annual IEEE/ACM international symposi um on Microarchitecture*, pages 330–335. IEEE Computer Society Press, 1997.
44. Dave Jagger, editor. *Advanced RISC Machines Architecture Reference Manual*. Prentice Hall, Jul 1995.
45. T. Austin, E. Larson, and D. Ernst. SimpleScalar: an infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, Feb 2002.
46. Steven J. E. Wilton and Norman P. Jouppi. CACTI: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, May 1996.