On the realizability of hardware microthreading. Revisiting the general-purpose processor interface: consequences and challenges

Poss, R.C.

Citation for published version (APA):
Chapter 9

Thread-local storage
—Highlight on specifics, opus 2

Abstract

Some functional requirements of operating software providers may be both invisible to hardware architects and invisible to application developers; they may exist to support the generality of the operating software or its reusability across platforms. From the hardware architect’s perspective, an opportunity exists to learn about these requirements and optimize a design accordingly. For example, Thread-Local Storage (TLS) is a necessary feature of run-time environments when threads must run general-purpose workloads. This is fully under the responsibility of the operating software, as TLS is “just memory” from the hardware provider’s perspective, and assumed to pre-exist by application developers. Meanwhile, on a massively concurrent architecture which encourages the creation of many short logical threads in hardware, such as the one introduced in part I, provisioning TLS via traditional heap allocators would be largely detrimental to performance and efficiency due to contention and over-allocation. In this chapter, we show the gains to be obtained by co-designing support for TLS between the hardware architect and the operating software provider. We provide an analysis of the requirements to provision TLS on such architectures and a review of the design space for solutions. We illustrate with our own implementation.

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1</td>
<td>Introduction</td>
<td>148</td>
</tr>
<tr>
<td>9.2</td>
<td>“Traditional” provisioning of TLS</td>
<td>148</td>
</tr>
<tr>
<td>9.3</td>
<td>Smarter provisioning of TLS</td>
<td>150</td>
</tr>
<tr>
<td>9.4</td>
<td>Implementation</td>
<td>151</td>
</tr>
<tr>
<td>9.5</td>
<td>Integration with the machine interface</td>
<td>159</td>
</tr>
<tr>
<td>Summary</td>
<td></td>
<td>160</td>
</tr>
</tbody>
</table>
CHAPTER 9. THREAD-LOCAL STORAGE

9.1 Introduction

A C compiler for most register-based machines uses a stack pointer for two purposes:

- to allocate and access function activation records\(^1\).
- when using the commonly-supported C extension “alloca” or [II99]/[II11b]’s variably sized arrays scoped to a function body, to perform the corresponding dynamic allocation at run-time.

In addition to this, most existing C compilers support the notion of “thread-local variables,” declared with a special storage qualifier (“thread” or similar), and which are guaranteed to designate, at run-time, a different object in each thread whose initial value in each thread is the initial value defined in the program at the point of definition. This definition has been captured in the latest specifications [II11a, II11b]. The implementation of this feature is based on a “thread local storage template” generated by the compiler statically, and passed by reference to each newly created thread to be duplicated in a thread-local memory.

Activation records, dynamic local allocation and thread-local variables define collectively the notion of “Thread-Local Storage (TLS),” which is an area of memory that can be used by thread program code with the assumption that no other thread will inadvertently overwrite its values. In general terms, TLS is required for a thread to properly virtualize a general-purpose computer with the computing abilities of a Turing machine (cf. section 1.2.1). Conversely, an execution environment must provision TLS to claim its support for general-purpose workloads.

In this chapter, we examine how TLS can be provisioned to threads efficiently in an architecture with fine-grained concurrency, and how access to TLS can be gained from the perspective of code generation.

9.2 “Traditional” provisioning of TLS

The two traditional strategies to obtain access to TLS in new threads are: \textit{dynamic pre-allocation}, before the thread starts by the creating thread, or \textit{self-allocation} by the newly created thread after it has started. With pre-allocation, a pointer to TLS can be passed as an argument, or at a predefined location. With self-allocation, a protocol must exist so that a thread can \textit{request} the allocation of TLS.

9.2.1 Dynamic pre-allocation

Dynamic pre-allocation requires \textit{a priori} knowledge of an upper bound on the TLS size. We found an extensive analysis of this in [Gru94]. To summarize, a combination of compiler-based analysis of the call graph in each thread program, together with a profile-directed analysis of the actual run-time requirements, can provide a conservative estimate of this upper bound in a large number of applications.

There are two issues with pre-allocation however. The first issue arises when a primitive exists to bulk create many logical threads over a limited set of hardware threads, such as presented in part I. Each logical thread potentially needs TLS. This has two consequences:

\(^1\) Activation records primarily contain local variables that do not fit in registers, e.g. local arrays, and also possibly spill space and the return address in called procedures.
since the mapping of logical threads to hardware thread contexts may not be visible in software, potentially as many separate TLS spaces would need to be allocated as there are logical threads in the family. This would cause severe over-allocation for large logical families running over a few execution units;
• any use of pre-allocation would require two values to be passed as argument (a base pointer and a size) and would mandate either a potentially expensive multiplication with the logical thread index in every thread, or sizes that are powers of two, to allow a cheaper shift operation instead.

The second issue is that pre-allocation is not an appropriate general scheme: it is unable to provide TLS for recursion where the depth cannot be statically bound, for example data-dependent recursions, and it cannot cater to dynamic allocation of local objects whose size is known only at run-time. Because of this latter issue, whether pre-allocation is used or not, a fully general system would thus need to also offer self-allocation as an option.

9.2.2 Self-allocation

Self-allocation, in contrast, seems relatively simple to implement. It suffices to emit, at the start of every thread program that requires TLS, an invocation of a system service in charge of dynamic memory allocation. Supposing such a service exists, we explore here how its interface should look. There are two possible strategies. Either the service is implemented by means of a regular procedure, which is called by branching control within the same thread, or it is implemented as a thread program, which is accessed by creating a new thread running that program. In both cases, some form of mutual exclusion over a shared memory allocator is required; otherwise, the two strategies differ as follows.

The procedure-based approach requires an allocation service that can be run from individual threads at a low cost. To keep logical thread creation lightweight, requirements such as a privilege switch or a trap to access the service would be inappropriate. The allocator would be a procedure that negotiates exclusive access to some data structure, determines a range of addresses to use as TLS, and releases the data structure. For best performance this should have a fixed, low overhead in most cases. In turn this requires minimizing non-local communication, hence requires a pooled memory allocator with per-processor pools. This is the approach taken by e.g. the “polo stack” allocation strategy from [Abd08, Chap. 6].

The thread-based approach creates the opportunity to delegate the allocation to some other resource on the system. This is beneficial as the designer may not wish to equip every processor with the hardware facilities required for a memory allocator, yet desire to allow every logical thread to access TLS. This is of course constrained by contention: the system must provision enough bandwidth to this service, either through faster processors implementing the service or a larger number thereof.

Then, the service interface must guarantee that the thread creation is always possible\(^2\). This in turn mandates that there is at least one thread context which is never automatically used by bulk creation except for TLS allocation requests, and that the program must be able to force the hardware “allocate” request to grab that specific context. This in turn requires a mechanism in hardware to reserve a context permanently and an addressing scheme to identify that context in the “allocate” operation.

\(^2\)The scheme described later in chapter 10 does not apply here as it pre-requires TLS to serialize the behavior locally.
To summarize, TLS self-allocation is a viable strategy which can either be implemented locally on each processor, or on service processors shared between multiple multithreaded cores. It requires mutual exclusion and an allocation algorithm with an extremely low overhead, so that thread creation stays cheap. Contention may be an issue when sharing the service between multiple cores, a topic that we revisit later in section 14.4.

9.3 Smarter provisioning of TLS

Dynamic pre-allocation and self-allocation require the time overhead of a procedure call or one extra thread creation and synchronization during the start-up of every logical thread. This overhead, commonly between 50 and 200 processor cycles on contemporary hardware (for procedure calls) or 50 cycles on the proposed architecture, does not compare favorably with the low logical thread activation overhead on the reference implementations of our new architecture (less than 5 processor cycles per logical thread). Moreover, since the machine interface does not guarantee that the values of the local synchronizers persist across logical threads sharing the same thread context, it would be necessary to either design a new architectural mechanism to carry the TLS pointers from one logical thread to the next, or invoke the allocation service in every logical thread, not only the first. We present two alternative strategies, presented below.

9.3.1 Persistent dynamic allocation

The first strategy we considered preserves a TLS range across logical threads sharing the same thread context. In this approach, each thread context retains, next to the program counter, a base pointer for the area of TLS reserved for all logical threads sharing the context. This is a valid approach since all logical threads execute in sequence without interleaving (sections 3.3.1 and 4.2). When the first logical thread in a family starts in a context, it checks this base pointer, and performs an initial allocation if the base pointer is not yet configured. All subsequent logical threads in the same context can reuse the same TLS.

Here there are two storage reclamation strategies: aggressive reclamation, where TLS is de-allocated as soon as the context is released, and delayed reclamation where TLS remains allocated between families.

To enable aggressive reclamation, either a hardware process, or the thread synchronizing on termination of a family, would explicitly de-allocate all the TLS areas allocated for each thread context effectively used by the family. This has linear time complexity with the number thread contexts used for the family per core, and is independent of the logical thread sequence which is typically larger. The cost of allocation is then repeated for each new created family.

With delayed reclamation, TLS space is reused across families: the cost of allocation is factored at the expense of possible over-allocation. There is a spectrum of implementation strategies. Asynchronous Garbage Collection (GC) of TLS areas for inactive thread contexts is possible if the application code guarantees no sharing of TLS data between threads. Otherwise, synchronous GC can be used, or the operating system can reclaim storage when the entire program terminates.

\[\text{We assume here that all threads have the same TLS size requirements. Otherwise, the size should be stored alongside the base pointer, compared upon thread startup, and the TLS should be reallocated when a new size is desired.}\]
9.4. IMPLEMENTATION

9.3.2 Context-based pre-reservation

We pushed the concept of persistent allocation with delayed reclamation further: we explored more static schemes where TLS space is pre-allocated for all thread contexts on a group of processors before a software component is assigned to this group.

To optimize this situation, we propose to allow a logical thread to construct a pointer to a region of TLS from the identity of the thread context where the logical thread runs. We experimented with multiple implementations, with various trade-offs between the latency benefit and the cost of the required extra logic. We determined two orthogonal design decisions: how the space is allocated from shared storage, and how the addresses are distributed to threads.

Regarding allocation, we found two implementable strategies:

- static pre-allocation: a predetermined amount of storage is allocated for each thread context. This is suitable if pre-allocation is possible, i.e. when there are known upper bounds on TLS size and the provision of TLS for all contexts fits in the available storage;
- deferred pre-reservation using virtual addressing: a predetermined number of virtual addresses are reserved for each thread context, then storage is allocated on-demand when the addresses in the TLS areas are first used. Although this scheme also bounds the amount of TLS via the number of addresses reserved, it is possible to make this bound so high that the addresses available to each thread context would be sufficient to address the entire physical storage.

Regarding distribution, we found two implementable strategies:

- external distribution: a table exists in the system, indexed by the logical processor identifier and thread context identifier of each thread, which can be configured to contain a different address and size for TLS for every thread context. This table can reside in main memory with a base address determined by convention, or in dedicated memory structures on chip next to each processor;
- computed distribution: a common base pointer and TLS size are accessible to multiple thread contexts by convention, and can be configured in software. In each thread, the program can compute a private area of TLS by multiplying a combination of its logical processor identifier and thread context identifier with the common size, and adding this to the common base pointer. The pair (base pointer, size) can be shared by all processors in the system, or configurable by processor.

We summarize the trade-offs of these four combinations in table 9.1.

The external/static and external/deferred schemes are the schemes used in most existing general-purpose operating systems to implement processes and threading. For example in GNU/Linux, Solaris and BSD each system thread is allocated a range of virtual addresses as stack, which is populated on demand while the thread runs. In uCLinux for embedded systems without virtual addressing, processes receive a dedicated subset of the physical memory as TLS.

9.4 Implementation

We did not implement dynamic pre-allocation in our technology (cf. chapter 6). We also did not thoroughly explore self-allocation or any scheme requiring mutual exclusion, including
### Thread-Local Storage

#### Distribution

<table>
<thead>
<tr>
<th>Allocation</th>
<th>External</th>
<th>Computed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>Thread contexts compete for a subset of the physically available memory, but heterogeneous sizes are possible. Time-expensive to pre-configure (linear in the number of contexts), cheap in logic (needs access only to the context identifier).</td>
<td>All thread contexts compete equally for a subset of the physically available memory. Cheapest in time and extra logic to pre-configure (constant time, needs access only to the context identifier), least flexible for software.</td>
</tr>
<tr>
<td>Deferred</td>
<td>Thread contexts compete for a subset of the virtual address space, and heterogeneous address windows are possible. Most time-expensive and logic-expensive to pre-configure (linear in the number of contexts, requires virtual addressing and handling of translation faults), most flexible for software.</td>
<td>All thread contexts compete equally for a subset of the virtual address space. Cheap in time to pre-configure (constant time), expensive in extra logic (requires virtual addressing).</td>
</tr>
</tbody>
</table>

Table 9.1: Trade-offs of context-based TLS partitioning

---

Side note 9.1: Support for heap-based dynamic allocation.

Although we did not study self-allocation for obtaining TLS space in threads, we did not deliberately avoid dynamic memory allocation altogether. Indeed, our software integration work eventually included a two stage allocator into the standard C library available to programs, to support the "`malloc`" API. This allocator handles small sizes locally using binned allocation of fixed sizes chunks in TLS-based "local heaps," and delegates allocation of larger sizes centrally by an off-the-shelf allocator using mutual exclusion, Doug Lea’s `dlmalloc` [Lea96]. Our allocator is available publicly with our SL software.

---

persistent dynamic allocation from section 9.3.1. We made this decision for two reasons. First, as highlighted at the start of section 9.3, the overhead of dynamic allocation is high compared to the latency of thread creation, and we were interested in exploring strategies that provide TLS at lower latencies. Then, the system implementation we had at our disposal did not provide facilities for mutual exclusion until a much later phase, where our techniques from section 9.3 were already implemented and demonstrated to perform adequately.

Considering the context-based pre-reservation schemes from section 9.3.2, we also avoided the schemes based on external distribution. We made this choice because we were focusing on parallel execution over large numbers of processors, and we predicted that external distribution would incur large pre-configuration costs every time a group of processors is recycled between benchmark applications.

Instead, during our work we explored the two remaining context-based schemes that we have identified: static/computed and deferred/computed. We found them attractive because they are simple to implement and the computed access to TLS makes the preconfiguration costs scalable to large numbers of processors and thread contexts.
9.4. IMPLEMENTATION

<table>
<thead>
<tr>
<th>Physical RAM reserved for TLS</th>
<th>Number of processors</th>
<th>Contexts per processor</th>
<th>TLS size</th>
</tr>
</thead>
<tbody>
<tr>
<td>256 KiB</td>
<td>1</td>
<td>64</td>
<td>4 KiB</td>
</tr>
<tr>
<td>16 MiB</td>
<td>1</td>
<td>64</td>
<td>256 KiB</td>
</tr>
<tr>
<td>256 KiB</td>
<td>8</td>
<td>8</td>
<td>4 KiB</td>
</tr>
<tr>
<td>16 MiB</td>
<td>8</td>
<td>8</td>
<td>256 KiB</td>
</tr>
<tr>
<td>1 GiB</td>
<td>8</td>
<td>8</td>
<td>16 MiB</td>
</tr>
<tr>
<td>256 KiB</td>
<td>64</td>
<td>64</td>
<td>64 B</td>
</tr>
<tr>
<td>16 MiB</td>
<td>64</td>
<td>64</td>
<td>4 KiB</td>
</tr>
<tr>
<td>1 GiB</td>
<td>64</td>
<td>64</td>
<td>256 KiB</td>
</tr>
</tbody>
</table>

Table 9.2: TLS size per thread with the static/computed scheme.

9.4.1 Static/computed partitioning

The first scheme we implemented was a context-based, static computed partitioning of the address space. In this scheme, the operating system allocates a fixed subset of the physical RAM and divides it equally across all thread contexts. This scheme does not allow the TLS space of a thread context to be expanded, with the same drawbacks as presented in section 9.2.1.

We present possible configurations in table 9.2. As can be seen in this table, the static/computed strategy does not scale well for larger numbers of processors and/or thread contexts per processor. It incurs significant wasted storage space, caused by different TLS requirements between logical threads: the common size must be large enough to cater to the highest requirement, which is typically uncommon at run-time. The amount of wasted space increases with the number of processors and/or the number of contexts per processors.

Nevertheless, this was the scheme eventually used to test the UTLEON3-based platform [DKK+12], as this implementation contains only 1 core.

9.4.2 Deferred/computed partitioning

The second scheme we implemented was context-based, deferred/computed partitioning of the address space with virtual addressing. We made the assumption that the entire system shares a single virtual address space, i.e. that the same address translation unit is used by all processors and the inter-processor memory network uses virtual addresses. Then we used the following address format for TLS:

```
MSB | LSB
---|---
1  | P | P | P | T | T | T | - | - | ... | -
```

In this scheme, a thread can construct a valid address for its TLS area by setting the Most Significant Bit (MSB), then concatenating its logical processor index and thread context index in the most significant bits of the address. Since this information is already present in each processor’s pipeline to support scheduling, this value can always be constructed cheaply (one or two pipeline cycles). The lower half of the address space (with the MSB unset) stays available for common data, code, shared heaps, system data structures, etc.

We document the maximum allowable TLS size per thread under various architecture configurations in table 9.3. Despite the static address space reservation, with 64-bit addressing there is sufficient address space capacity in each thread to potentially address the
Table 9.3: Maximum possible TLS sizes with the deferred/computed scheme.

<table>
<thead>
<tr>
<th>Number of address bits</th>
<th>Number of processors</th>
<th>Contexts per processor</th>
<th>Maximum size</th>
<th>TLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>8</td>
<td>8</td>
<td>32 MiB</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>64</td>
<td>64</td>
<td>512 KiB</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>256</td>
<td>8 KiB</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>64</td>
<td>2 EiB</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>1024</td>
<td>256</td>
<td>32 PiB</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>1024</td>
<td>1024</td>
<td>8 PiB</td>
<td></td>
</tr>
</tbody>
</table>

Figure 9.1: Potential bank and line conflicts with 64-bit addresses, 1024 cores, 256 contexts per core.

entire capacity of contemporary physical RAM chips. With 32-bit addressing however, this scheme is restrictive except with very few thread contexts overall.

While this scheme is transparent to use in programs, we found three significant issues at the level of system integration: cache and bank conflicts, Translation Lookaside Buffer (TLB) pressure and storage reclamation.

### 9.4.2.1 Cache and bank conflicts

Since most threads use only a small part of their TLS area, from the perspective of the memory system, most accesses are made to addresses that differ only via their most and least significant bits. Meanwhile, most common designs for bank and cache line selection in hardware select banks based on a tag computed with the middle bits of the address, which will be mostly identical between all threads in our scheme (fig. 9.1).

This effect was recognized in previous work [MS09], but existing solutions to avoid this situation are mostly applicable to external TLS distribution schemes, e.g. by randomization of the TLS base address for each thread context. This cannot be used here since we use computed distributions.

Instead, cache and bank selection randomization techniques do apply and are effective at mitigating this effect [RH90, Rau91, Sez93, GVTP97, KISL04, VD05].

To test this effect we have used four synthetic multithreaded benchmarks, two which do not use TLS at all and two that use less than 512 bytes of TLS per thread. The two programs
which do not use TLS perform simple data-parallel computations. The two programs using TLS perform a more complex data-parallel operation which requires several procedure calls, and thus multiple stack-related operations besides the computation. We then ran these programs on a single multithreaded core with various numbers of thread contexts, given in the figures’ x-axes. The processor is connected to a 128KiB 4-way associative L2-cache, which fits the working set of all 4 benchmarks.

As can be seen in fig. 9.2, the pattern of memory accesses in the two latter benchmarks incur a per-thread latency of several hundred cycles, contrasting with less than 50 cycles per thread for the simple benchmarks. Despite the potential for latency tolerance offered by fine-grained hardware multithreading, the number of instructions per cycle remains low (<50%, fig. 9.2b). Monitoring of the L2 cache activity reveals that more than 40% of accesses are misses caused by conflicts.

We then implemented an XOR reduction of the top and lower address bits to map L2 cache lines, inspired from [GVTP97]. As can be seen in fig. 9.3, this simple change caused a 400% performance increase of the most memory-bound benchmark, mostly by allowing independent memory accesses by different threads and thereby increasing IPC (fig. 9.3b).
CHAPTER 9. THREAD-LOCAL STORAGE

Figure 9.4: Address bit shuffling for non-TLS pages.

Figure 9.5: Address bit shuffling to aggregate distinct small TLS spaces into shared physical pages.

Note that the IPC measurement anomaly in fig. 9.3b is caused by an imprecision in our accounting: some cycles attributed to one benchmark, which causes its IPC to exceed 100%, should be accounted for another instead.

To summarize, although an extensive study of cache design was not within the scope of our research, our work highlights that this TLS scheme places a strong need for tag randomization in the cache system.

9.4.2.2 TLB pressure

With a naive implementation of virtual addressing, our scheme would require at least one different translation entry for every thread context, because the address spaces of each TLS area (megabytes for 32-bit addressing, petabytes for 64-bit addressing, cf. table 9.3) would be larger than the virtual addressing page size (typically kilobytes on current systems). With large numbers of processors and thread contexts (e.g. 256×1024 contexts on large chips), this would require inefficiently large TLB implementations.

However, considering that the actual required TLS space in each thread is typically smaller than the page size, e.g. a few hundred bytes if only a few activation records are defined, we can propose an optimization to aggregate the actually used regions of TLS within single translation entries. To do this, we organize virtual addressing as follows.

We consider that there are \(2^P\) cores (e.g. \(P = 6\), 64 cores) and \(2^T\) thread contexts per core (e.g. \(C = 7\), 128 contexts). We also consider that there are \(2^C\) bytes in a cache line (e.g. \(C = 6\), 64 bytes). As a first step, we require that the virtual page size is at least \(2^{C+P+T}\) bytes wide (e.g. 512KiB in our example). We consider that each page contains \(2^B\) bytes,
### 9.4. IMPLEMENTATION

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Contexts per processor</th>
<th>Page size</th>
<th>TLS storage per thread in each page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>4 KiB</td>
<td>64 B</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>16 KiB</td>
<td>256 B</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1 MiB</td>
<td>16 KiB</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>256 KiB</td>
<td>64 B</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>1 MiB</td>
<td>256 B</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>16 MiB</td>
<td>4 KiB</td>
</tr>
<tr>
<td>1024</td>
<td>256</td>
<td>16 MiB</td>
<td>64 B</td>
</tr>
<tr>
<td>1024</td>
<td>256</td>
<td>128 MiB</td>
<td>512 B</td>
</tr>
<tr>
<td>1024</td>
<td>256</td>
<td>1 GiB</td>
<td>4 KiB</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>64 MiB</td>
<td>64 B</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>1 GiB</td>
<td>1 KiB</td>
</tr>
<tr>
<td>1024</td>
<td>1024</td>
<td>16 GiB</td>
<td>16 KiB</td>
</tr>
</tbody>
</table>

Table 9.4: Amount of TLS storage provided for each thread per virtual page.

Values assuming 64-byte cache lines.

with \( B \geq C + P + T \). Then we modify the Memory Management Unit (MMU) as follows. When a request to translate a virtual address enters the MMU, the highest address bit is tested. If the bit is set, indicating TLS, the virtual address is first translated as depicted in fig. 9.5:

1. the lowest \( B - P - T \) bits are left unchanged;
2. the \( P + T \) most significant bits (except the highest) are shifted next to the lowest \( B - P - T \) bits forming a complete in-page address;
3. the most significant bit (set) is shifted as the least significant bit of the page address;
4. the remaining original bits (from position \( B - P - T + 1 \) to \( B - 1 \)) are shifted to the remaining positions of the page address (from \( B + 2 \) onwards).

If the highest bit is not set, then the address is first translated as depicted in fig. 9.4:

1. the lowest \( B \) bits are left unchanged;
2. the most significant bit (unset) is shifted as the least significant bit of the page address;
3. the remaining original bits (from position \( B + 1 \) onwards) are shifted to the remaining positions of the page address (from \( B + 2 \) onwards).

(The lowest \( C \) address bits are left unchanged to preserve locality within cache lines.) Then the resulting page address is looked up in the TLB and/or translation table(s).

With this scheme in place, each virtual page with an odd address provides \( 2^{B-T-P} \) bytes of storage to every thread context in the system (e.g. 64 bytes with 512KiB pages, 2KiB with 16MiB pages). This way, the number of TLB entries grows only as a function of the largest TLS address effectively used in the system. Since it is uncommon to see a large number of threads using simultaneously large TLS addresses, this relieves pressure on the TLB.

We provide a few examples of the provision for TLS per virtual page in table 9.4. As can be seen in this table, usual page sizes on contemporary operating systems (4KiB to 16MiB) are sufficient to provision at least a cache line per thread context up to moderately sized system (64 cores), and only a four-fold increase to at least 64 MiB per page is sufficient to support future-generation chips with thousands of cores.
9.4.2.3 Reclamation of TLS storage

To analyze the issues related to reclamation, we must separately consider the situation without the TLB aggregation scheme presented above, and the situation once it is enabled.

Without the optimization, the issues of reclamation for this scheme are similar to those from the persistent dynamic allocation scheme described in section 9.3.1. The main difference is that the TLS address ranges are defined as translation entries in the virtual addressing subsystem, instead of per-context base addresses on each processor.

This makes aggressive reclamation difficult to implement: since the program and the virtual addressing subsystem are typically isolated from each other, the program must signal to an operating system every bulk creation and termination event to enable reclamation. Since the overhead of interactions with system services is typically higher than local procedure calls, it would defeat the entire approach and make persistent dynamic allocation more attractive. Context-based partitioning with deferred allocation and aggressive reclamation would only be viable if virtual addressing is under direct control of the program.

Delayed reclamation, in contrast, is tractable: a concurrent GC can asynchronously inspect the areas of memory pointed to by the active translation entries, and unmapped all the unused areas. Or, alternatively, an operating system can release all translation entries at once when the program terminates.

With the aggregation optimization presented in the previous section, there is a significant problem however: \textit{reclamation becomes impossible because each page contains a part of the TLS area of all thread contexts in the system} due to the bit shuffling.

To overcome this obstacle, we introduce \textit{virtual resource identifiers} as part of the addressing scheme in addition to the physical processor identifiers, as depicted in fig. 9.6. These are numerical identifiers that delimit a memory management domain, for example a process in an operating system. In the addressing scheme, each access to TLS by a thread places the virtual resource identifier above the processor bits in virtual addresses, and the MMU shifts them to form the Least Significant Bit (LSB) of page addresses prior to address translation.

We suggest that this extension only applies to implementations with 64-bit addressing, since the size of the virtual resource identifier constrains the address width usable to address memory. Also, the width of the virtual address identifier in bits directly impacts TLB pressure, since TLS addresses from threads with different resource identifiers will map to different pages. This is similar to how synonyms increase cache pressure in virtually tagged caches [CD97a, CD97b].
However, virtual resource identifiers solve the reclamation problem. Indeed, since the resource identifier is part of the page identifier, two application components using distinct resource identifiers will map to different pages, and reclamation becomes possible by reclaiming entire sets of pages sharing the same resource identifier. This assumes that applications are constrained to avoid sharing of memory between program components that have distinct resource identifiers, or that explicit registration of common memory objects to a reclamation system service is mandatory.

Final reclamation at program termination is the reclamation strategy we adopted in our implementation. We expect concurrent GC of translation entries to be better suited against over-allocation in long-running programs, however we could not explore this direction due to our limited effort budget.

9.5 Integration with the machine interface

Depending on the selected TLS scheme for a given implementation, the required architecture support differs:

- for dynamic pre-allocation and self-allocation, some form of mutual exclusion is required;
- for persistent dynamic allocation, both mutual exclusion and access to a per-context TLS base pointer and size is required;
- for all context-based partitioning schemes using external distribution, mutual exclusion may not be required, but access to the per-context TLS pointer and TLS size is required;
- for all context-based partitioning schemes using computed distribution, mutual exclusion may not be required but access to a common TLS base pointer and TLS address space size are required to compute the actual local, per-context TLS base pointer.

To summarize, either mutual exclusion or a way to obtain or configure the per-context TLS base pointer and size is required, or both. Since mutual exclusion can be introduced by traditional means (either memory-based atomics or as an operating system service), we focus here on the latter.

The requirement is for each logical thread to be able to retrieve the TLS base pointer for its own thread context, as well as the corresponding TLS size. Alternatively, the thread can be allowed to retrieve the first and last address of its TLS area, from which it can compute the size if needed.

In our work, we opted for the following extension to the machine interface:

- two new operations “ldbp” and “ldfp” are introduced, with the corresponding assembly mnemonics “ldbp” and “ldfp.” The first loads the base TLS address into its target operand, and the second loads the first address past the end of the TLS area. This allows us to both derive the size of the TLS and let a system implementation choose for upwards or downwards stack growth;
- in addition, two new operations “gettid” and “getcid” are introduced, with the corresponding mnemonics “gettid” and “getcid.” The first loads the local thread context identifier of the issuing thread into its target operand, and the second loads the local processor (core) identifier. Both use 0-based indexing. This allows us to implement context-based TLS schemes using explicit distribution with arbitrary data structures in memory.
9.5.1 Example with the 1-core FPGA prototype

On the UTLEON3-based FPGA prototype implementation, we used the static/computed scheme, with 1 KiB bytes of RAM allocated to each of the 128 thread contexts.

To implement this, a 128 KiB array is defined in the program data. A linker symbol named "\_\_first\_tls\_top" is placed at the address 1 KiB past the start of the array in memory. The compiler then emits the sequence in listing 9.1 at the start of every thread program which uses TLS. At run-time, each of these instructions executes in one processor cycle and access to TLS is gained in 5 cycles. It is possible to reduce this to 4 cycles if we fix the size to a static value, in which case the first operation is not needed and the size can be given as immediate constant to "sll."

9.5.2 Example with the multi-core system emulation

On the MGSim system emulation, we used the deferred/computed scheme. Here the implementation is simpler: the compiler simply needs to emit a single use of "ldfp $sp" at the start of every thread program which uses TLS. At run-time, the bits required to construct the TLS final address are available in the pipeline and allow the operation to complete in 1 processor cycle. The size of the individual TLS areas can be further obtained by subtracting the output of "ldbp" from the output of "ldfp."

Listing 9.1: Code sequence to access TLS on UTLEON3.
### 9.5. INTEGRATION WITH THE MACHINE INTERFACE

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Mechanism</th>
<th>Time cost†</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic pre-allocation</td>
<td>procedure call or active message by creating thread</td>
<td>$A_1$</td>
</tr>
<tr>
<td>Self-allocation</td>
<td>procedure call or active message by each created logical thread</td>
<td>$A_1' \times N$</td>
</tr>
<tr>
<td>Persistent dynamic allocation</td>
<td>procedure call or active message by the first logical thread of a context</td>
<td>$A_1'' \times P$</td>
</tr>
<tr>
<td>Context-based pre-reservation</td>
<td>static pre-allocation, external distribution</td>
<td>$A_1 + A_2 + C_1(P)$</td>
</tr>
<tr>
<td>Context-based pre-reservation</td>
<td>deferred pre-allocation, external distribution</td>
<td>$A_2 + C_1(P) + C_2(P)$</td>
</tr>
<tr>
<td>Context-based pre-reservation</td>
<td>static pre-allocation, computed distribution</td>
<td>$A_1 + C_3(N)$</td>
</tr>
<tr>
<td>Context-based pre-reservation</td>
<td>deferred pre-allocation, computed distribution</td>
<td>$C_4(P,N)$</td>
</tr>
</tbody>
</table>

† $N$: number of logical threads created. $P$: number of thread contexts participating in the bulk creation. $A_1$: cost to perform one allocation of TLS space. $A_2$: cost to allocate an array of pointers. $C_1(P)$: cost to compute and store one pointer for each context. $C_2(P)$: cost to provision virtual memory upon first access by each context (may be aggregated across multiple contexts and occur concurrently). $C_3(N)$: cost to compute one pointer in each logical thread (may occur concurrently). $C_4(P,N)$: combined cost of provisioning virtual memory and computing pointers (may occur concurrently).

Table 9.5: Protocols to provision TLS for one bulk creation.

### Summary

Provisioning Thread-Local Storage (TLS) to threads is an essential feature of a general-purpose system, mostly because per-thread stacks are necessary to support recursion and arbitrary large numbers of local variables.

One specific challenge related to massively concurrent architectures is support for a large number of threads and the design objective to keep thread creation cheap throughout the abstraction stack. This opposes traditional operating software approaches which exploit a common memory allocation service using mutual exclusion, as this would become a contention point as the rate of thread creation increases overall.

Another challenge related to the specific design from part I is that TLS is a feature of logical threads, whereas our physical unit of concurrency is the physical thread context. As the number of logical threads can be arbitrary large but they are serialized over the hardware contexts, TLS should be allocated per context and not per logical thread.

In this chapter we have reviewed traditional approaches to provision TLS to threads. We summarize the various options in table 9.5. We presented multiple schemes that pre-allocate address space and associate semi-statically TLS areas to thread contexts. They address both challenges identified above at once. They involve joint support from the architecture and from operating software providers, in particular with regards to reclamation of unused TLS ranges.