ELF Handling For Thread-Local Storage

Ulrich Drepper, Red Hat Inc.
`drepper@redhat.com`

Version 0.20
February 8, 2003

Based on:

Intel Itanium Processorspecific Application Binary Interface, May 2001, Document Number: 245370-003

Thread-Local Storage, Version 0.68, Sun Microsystems, September 4, 2001

1 Motivation

Increasing use of threads lead developers to wish for a better way of dealing with thread-local data. The POSIX thread interface defines interfaces which allow storing `void *` objects separate for each thread. But the interface is cumbersome to use. A key for the object has to be allocated dynamically at run-time. If the key isn't used anymore it must be freed. While this is already a lot of work and error prone it becomes a real problem when combined with dynamically loaded code.

To counter these problems it was decided to extend the programming languages to let the compiler take over the job. For C and C++ the new keyword `__thread` can be used in variable definitions and declarations. This is not an official extension of the language but compiler writers are encouraged to implement them to support the new ABI. Variables defined and declared this way would automatically be allocated local to each thread:

```
__thread int i;
__thread struct state s;
extern __thread char *p;
```

The usefulness of this is not limited to user-programs. The run-time environment can also take advantage of it (e.g., the global variable `errno` must be thread-local) and compilers can perform optimizations which create non-automatic variables. Note that adding `__thread` to the definition of an automatic variable makes no sense and is not allowed since automatic variables are always thread-local. Static function-scope variables on the other hands are candidates, though.

The thread-local variables behave as expected. The address operator returns the address of the variable for the current thread. The memory allocated for thread-local variables in dynamically loaded modules gets freed if the module is unloaded. The only real limitation is that in C++ programs thread-local variables must not require a static constructor.

To implement this new feature the run-time environment must be changed. The binary format must be extended to define thread-local variables separate from normal variables. The dynamic loader must be able to initialize these special data sections.

The thread library must be changed to allocate new thread-local data sections for new threads. The rest of this document will describe the changes to ELF format and what the run-time environment has to do.

Not all architectures ELF is available for are supported in the moment. The list of architectures which is supported and described in this document are:

- IA-32
- IA-64
- SPARC (32-bit and 64-bit)
- SuperHitachi (SH)
- Alpha
- x86-64
- S390 (31-bit and 64-bit)

The description for HP/PA 64-bit awaits integration into this document and all other architectures have as of the time of this writing no (finalized) support.

Table 1: Section table entries for `.tbss` and `.tdata`

Field	.tbss	.tdata
`sh_name`	`.tbss`	`.tdata`
`sh_type`	`SHT_NOBITS`	`SHT_PROGBITS`
`sh_flags`	`SHF_ALLOC + SHF_WRITE + SHF_TLS`	`SHF_ALLOC + SHF_WRITE + SHF_TLS`
`sh_addr`	virtual address of section	virtual address of section
`sh_offset`	0	file offset of initialization image
`sh_size`	size of section	size of section
`sh_link`	`SHN_UNDEF`	`SHN_UNDEF`
`sh_info`	0	0
`sh_addralign`	alignment of section	alignment of section
`sh_entsize`	0	0

2 Data Definitions

The changes required to emit thread-local data objects are minimal. Instead of putting variables in sections `.data` and `.bss` for initialized and uninitialized data respectively, thread-local variables are found in `.tdata` and `.tbss`. These sections are defined just like the non-threaded counterparts with just one more flag set in the flags for the section. The section table entries for these sections look as shown in table 1. As can be seen the only difference to a normal data section is that the `SHF_TLS` flag is set.

The names of the sections, as is in theory the case for all sections in ELF files, are not important. Instead the linker will treat all sections of type `SHT_PROGBITS` with the `SHF_TLS` flags set as `.tdata` sections, and all sections of type `SHT_NOBITS` with `SHF_TLS` set as `.tbss` sections. It is the responsibility of the producer of the input files to make sure the other fields are compatible with what is described in table 1.

Unlike the normal `.data` sections the running program will not use the `.tdata` section directly. The section is possibly modified at startup time by the dynamic linker performing relocations but after that the section data is kept around as the **initialization image** and not modified anymore. For each thread, including the initial one, new memory is allocated into which then the content of the initialization image is copied. This ensures that all threads get the same starting conditions.

Since there is no one address associated with any symbol for a thread-local variable the normally used symbol table entries cannot be used. In executables the `st_value` field would contain the absolute address of the variable at run-time, in DSOs the value would be relative to the load address. Neither is viable for TLS variables. For this reason a new symbol type `STT_TLS` is introduced. Entries of this type are created for all symbols referring to thread-local storage. In object files the `st_value` field would contain the usual offset from the beginning of the section the `st_shndx` field refers to. For executables and DSOs the `st_value` field contains the offset of the variable in the TLS initialization image.

Table 3: Program header table entry for initialization image

Field	**Value**
`p_type`	`PT_TLS`
`p_offset`	File offset of the TLS initialization image
`p_vaddr`	Virtual memory address of the TLS initialization image
`p_paddr`	Reserved
`p_filesz`	Size of the TLS initialization image
`p_memsz`	Total size of the TLS template
`p_flags`	`PF_R`
`p_align`	Alignment of the TLS template

The only relocations which are allowed to use symbols of type `STT_TLS` are those which are introduced for handling TLS. These relocations cannot use symbols of any other type.

To allow the dynamic linker to perform this initialization the position of the initialization image must be known at run-time. The section header is not usable; instead a new program header entry is created. The content is as specified in table 3.

Beside the program header entry the only other information the dynamic linker needs is the `DF_STATIC_TLS` flag in the `DT_FLAGS` entry in the dynamic section. This flag allows to reject loading modules dynamically which are created with the static model. The next section will introduce these two models.

Each thread-local variable is identified by an offset from the beginning of the thread-local storage section (in memory, the `.tbss` section is allocated directly following the `.tdata` section, with the aligment obeyed). No virtual address can be computed at link-time, not even for executables which otherwise are completely relocated.

3 Run-Time Handling of TLS

As mentioned above, the handling of thread-local storage is not as simple as that of normal data. The data sections cannot simply be made available to the process and then used. Instead multiple copies must be created, all initialized from the same initialization image.

In addition the run-time support should avoid creating the thread-local storage if it is not necessary. For instance, a loaded module might only be used by one thread of the many which make up the process. It would be a waste of memory and time to allocate the storage for all threads. A lazy method is wanted. This is not much extra burden since the requirement to handle dynamically loaded objects already requires recognizing storage which is not yet allocated. This is the only alternative to stopping all threads and allocating storage for all threads before letting them run again.

We will see that for performance reasons it is not always possible to use the lazy allocation of thread-local storage. At least the thread-local storage for the application itself and the initially loaded DSOs are usually always allocated right away.

With the allocation of the memory the problems with using thread-local storage are not yet over. The symbol lookup rules the ELF binary format defines do not allow to determine the object which contains the used definition at link-time. And if the object is not known the offset of the variable inside the thread-local storage section for the object cannot be determine either. Therefore the normal linking process cannot happen.

A thread-local variable is therefore identified by a reference to the object (and therefore thread-local storage section of the object) and the offset of the variable in the thread-local storage section. To map these values to actual virtual addresses the run-time needs some data structures which did not exist so far. They must allow to map the object reference to an address for the respective thread-local storage section of the module for the current thread. For this two variants are currently defined. The specifics of the ABIs for different architectures require two variants.[1]

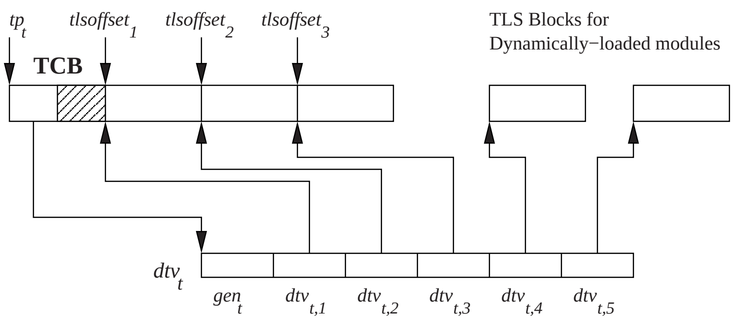


Figure 1: Thread-local storage data structures, variant I

[1] One reason to use variant II is that for historic reasons the layout of the memory pointed to by the thread register is incompatible with variant I.

Variant I (see figure 1) for the thread-local storage data structures were developed as part of the IA-64 ABI. Being brand-new, compatibility was no issue. The thread register for thread t is denoted by tp_t. It points to a Thread Control Block (TCB) which contains at offset zero a pointer to the dynamic thread vector dtv_t for the thread.

The dynamic thread vector contains in the first field a generation number gen_t which is used in the deferred resizing of the dtv_t and allocation of TLS blocks described below. The other fields contain pointers to the TLS blocks for the various modules loaded. The TLS blocks for the modules loaded at startup time are located directly following the TCB and therefore have an architecture-specific, fixed offset from the address of the thread pointer. For all initially available modules the offset of any TLS block (and therefore thread-local variable) from the TCB must be fixed after the program start.

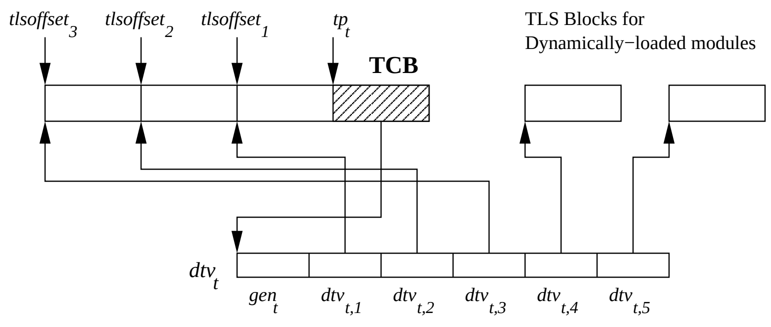


Figure 2: Thread-local storage data structures, variant II

Variant II has a similar structure. The only difference is that the thread pointer points to a Thread Control Block of unspecified size and content. Somewhere the TCB contains a pointer to the dynamic thread vector but it is not specified where. This is under control of the run-time environment and the pointer must not be assumed to be directly accessible; compilers are not allowed to emit code which directly access the dtv_t.

The TLS blocks for the executable itself and all the modules loaded at startup are located just below the address the thread pointer points to. This allows compilers to emit code which directly accesses this memory. Access to the TLS blocks is possible again through the dynamic thread vector, which has the same structure as in variant I, but also relative to the thread pointer with some offset which is fixed after the program starts. The offset of TLS data for the executable itself is even known at link-time.

At program start time the TCB along with the dynamic thread vector is created for the main thread. The position of the TLS blocks for the individual modules is computed using architecture specific formulas based on the size and alignment requirements ($tlssize_x$ and $align_x$) of the respective TLS block. In the architecture specific sections the formulas will use a function ‘round’ which returns its first argument rounded up to the next multiple of its second argument:

$$\text{round}(x, y) \quad = \quad y \times \lceil x/y \rceil$$

The memory for the TLS blocks does not necessarily has to be allocated right away. It depends on the model, static or dynamic, the module is compiled with whether it is necessary or not. If the static model is used the address (better said, offset from the thread pointer tp_t) is computed using relocations by the dynamic linker at program start time and compiler generated code directly uses these offsets to find the variable addresses. In this case memory has to be allocated right away. In the dynamic model finding the address of a variable is deferred to a function named `__tls_get_addr` which is provided by the run-time environment. This function is also able to allocate and initialize the necessary memory if this has not happened yet.

3.1 Startup and Later

For programs using thread-local storage the startup code must set up the memory for the initial thread before transferring control. Support for thread-local storage in statically linked applications is limited. Some platforms (like IA-64) don't define static linking in the ABI (if it is supported it is non-standard), other platforms like Sun's discourage the use of static linking since only limited functionality is available. In any case is dynamically loading modules in statically linked code severely limited or completely impossible. Therefore is the handling of thread-local storage very much simpler since only one module, the executable itself, exists.

The more interesting case is handling thread-local-storage in dynamically linked code. In this case the dynamic linker must include support for handling this kind of data sections. The requirements added by the ability to dynamically load code which uses thread-local storage are described in the next section.

To set up the memory for the thread-local storage the dynamic linker gets the information about each module's thread-local storage requirements from the `PT_TLS` program header entry (see table 3). The information of all modules is collected. This can possibly be handled with a linked list of records which contain

- a pointer to the TLS initialization image,
- the size of the TLS initialization image,
- the $\mathit{tlsoffset}_m$ for the module,
- a flag indicating whether the module uses the static TLS model (only if the architecture supports the static TLS model).

This list will be extended when dynamically loading additional modules (see next section) and it will be used by the thread library to set up the TLS blocks for a newly created thread. It would also be possible to merge two or more initialization records for the initial set of modules to shorten the list.

If all TLS memory would have to be allocated at startup time the total size would be $\mathit{tlssize}_S = \mathit{tlsoffset}_M + \mathit{tlssize}_M$ where M is the number of modules present at startup time. It is not necessary to allocated all this memory right away unless one module is compiled for the static model. If all modules use the dynamic model it is possible to defer the allocation. An optimized implementation will not blindly follow the flag indicating the use of the static model. If the required amount of memory is small it might not be worth the effort to defer the allocation, it might even save time and resources.

As explained at the beginning of this section, a variable in thread-local storage is specified by a reference to a module and an offset in the TLS block. Given the dynamic thread vector data structure we can define the module reference as an integer starting with 1 (one) which can be used to index the dtv_t array. The number each module receives is up to the run-time environment. Only the executable itself must receive a fixed number, 1 (one), and all other loaded modules must have different numbers.

Computing the thread-specific address of a TLS variable is therefore a simple operation which can be performed by compiler-generated code which uses variant I. But it cannot be done by the compiler for architectures following variant II and there is also a good reason to not do it: deferred allocation (see below).

Instead a function named `__tls_get_addr` is defined which could in theory be implemented like this (this is the form this function has for IA-64; other architectures might use a different interface):

```
void *
__tls_get_addr (size_t m, size_t offset)
{
  char *tls_block = dtv[thread_id][m];

  return tls_block + offset;
}
```

How the vector `dtv[thread_id]` is located is architecture specific. The sections describing the architecture-dependent parts of the ABIs will give some examples. One should regard the expression `dtv[thread_id]` as a symbolic representation of this process. `m` is the module ID, assigned by the dynamic linker at the time the module (application itself or a DSO) was loaded.

Using the `__tls_get_addr` function has the additional advantage to allow implementing the dynamic model where the allocation of the TLS blocks is deferred to the first use. For this we simply have to fill the `dtv[thread_id]` vector with a special value which can be distinguished from any regular value and possibly the value indicating an empty entry. It is simple to change the implementation of `__tls_get_addr` to do the extra work:

```
void *
__tls_get_addr (size_t m, size_t offset)
{
  char *tls_block = dtv[thread_id][m];

  if (tls_block == UNALLOCATED_TLS_BLOCK)
    tls_block = dtv[thread_id][m] = allocate_tls (m);

  return tls_block + offset;
}
```

The function `allocate_tls` needs to determine the memory requirements for the TLS of module `m` and initialize it appropriately. As described in section 2 there are two kinds of data: initialized and uninitialized. The initialized data must be copied from the relocated initialization image set up when module `m` was loaded. The uninitialized data must be set to zero. An implementation could look like this:

```
void *
allocate_tls (size_t m)
{
  void *mem = malloc (tlssize[m]);
  memset (mempcpy (mem, tlsinit_image[m], tlsinit_size[m]),
          '\0', tlssize[m] - tlsinit_size[m]);
  return mem;
}
```

`tlssize[m]`, `tlsinit_size[m]`, and `tlsinit_image[m]` have to be determined in an implementation-dependent way. They are all known after module `m` has been loaded. Note that the same image `tlsinit_image[m]` is used for all threads, whenever they are created. A thread does not inherit the data from it's parent.

Both variants for the storage data structures allow using the static model. The modules which are compiled this way can be recognized by the `DF_STATIC_TLS` flag in the `DT_FLAGS` entry in the dynamic section. If such a module is part of the initial set of modules (remember, such modules cannot be loaded dynamically) the memory for the TLS block must be allocated immediately at startup time for the initial thread and whenever a new thread is created for this new thread. Otherwise the allocation can be deferred and the elements of dtv_t are set to an implementation defined value (`UNALLOCATED_TLS_BLOCK` in the example code above).

3.2 Dynamic Loading

Dynamic loading of modules adds some more complexity to the picture. First, there should not be a limit on how many modules which use thread-local storage can be loaded at one point which means the dtv_t arrays must be enlarged if necessary. Second, it is absolutely necessary to avoid memory leaks. This must be kept in mind when optimizing the implementation for speed. The speed problems arise when deallocating memory of the TLS block of an unloaded module. The slots in the dynamic thread vector must be reused sooner or later. Not doing this would mean constantly extending the vector when loading new modules.

Since deallocating and then reallocating memory is expensive, especially since it has to be done for each individual thread, one might want to avoid the costs by keeping the memory around. But this must never lead to memory leaks if the same module is loaded and unloaded multiple times.

Now that the restrictions of the implementation are clear the actual work which has to be performed must be described. Dynamically loading modules which contain thread-local storage requires preparing the application for using the currently running and all future threads for using this memory. Note that loading modules which do not use thread-local storage themselves do not require special attention regardless of whether the rest of the program uses thread-local storage. The information about the new TLS block must be added to the list of initialization records and the counter for the number of loaded modules M must be incremented. While this takes care of threads which will be created afterwards already running threads must be prepared, too.

Loading a new module can mean that the size of the dynamic thread vector allocated for any given thread is possibly too small. This is what the generation counter gen_t in each dtv_t helps to detect. If the vector is accessed the first thing to do is to make sure the generation number is up-to-date and if not, allocate a larger vector. While this theoretically could be done by the thread which creates the new thread (or the new thread itself) this would only lead to sychronization problems and possibly unnecessary work if a thread does not use any thread-local storage. Since dynamically loaded modules cannot use the static model it is never necessary to allocate new elements in dtv_t right away. It is always possible to defer this until the first use in which case `__tls_get_addr` is used.

3.3 Statically Linked Applications

The TLS handling in statically linked applications is much simpler than in dynamically linked code. At least if it is determined that statically linked applications cannot dynamically load more modules. Even on systems which under some circumstances allow dynamically loading (such as systems using the GNU C library) dynamic loading might be restricted to loading to very basic modules and disallow those modules containing code using or defining thread-local storage.

Therefore statically linked code always has exactly one TLS block. And since only one module is ever used there is also no question about the variable offsets. Since all thread-local variables must be contained in this one TLS block the offset is also known at link-time.

The linker will always be able to fill in the module ID and offset and perform code relaxations. There is no work for the startup code to except setting up the TLS block for the initial thread. The thread library will have to do the same for newly created threads. This is a simple task since there is exactly one initialization image.

From the discussions in this section we can already see that the access of the TLS blocks is very simple since the $tlsoffset_1$ value is known at link-time and adding the thread pointer, the $tlsoffset_1$ value, and the variable offset results in the address of the variable. For some architectures the linker can automatically help to improve the code by rewriting the compiler-generated code. When discussing the thread-local storage access models we will see how much simpler the code gets and when discussing the linker relaxations we will see how the linker can perform all the necessary optimizations.

3.4 Architecture Specific Definitions

Not all architectures use the same variant for the thread-local storage data structures and some other requirements are also different. The handling of the thread pointer is so low-level that it naturally is architecture specific. This section describes these bits to fill in the gaps in the discussion so far and prepares for the description of the inner workings of the startup code.

3.4.1 IA-64 specific

The IA-64 ABI specifies the use of thread-local storage data structures according to variant I above. The size of the TCB is 16 bytes where the first 8 bytes contain the pointer to the dynamic thread vector. The other 8 bytes are reserved for the implementation.

The address of the dtv_t array can be determined by loading the 64-bit word tp_t pointed to by the thread register, `tp` (GR 13). Each element of dtv_t is 8 bytes in size to accommodate a pointer.

The TLS blocks for all modules present at startup time (i.e. those which cannot be unloaded) are created consecutively following the TCB. The $\mathit{tlsoffset}_x$ values are computed as follows:

$$\begin{aligned}\mathit{tlsoffset}_1 &= \mathrm{round}(16, \mathit{align}_1)\\ \mathit{tlsoffset}_{m+1} &= \mathrm{round}(\mathit{tlsoffset}_m + \mathit{tlssize}_m, \mathit{align}_{m+1})\end{aligned}$$

for all m in $1 \leq m \leq M$ where M is the total number of modules.

The function `__tls_get_addr` is defined in the IA-64 ABI as described above:

```
extern void *__tls_get_addr (size_t m, size_t offset);
```

It takes the module ID and the offset as parameters requiring relocations to change the calling code to provide the needed information.

3.4.2 IA-32 specific

The IA-32 ABIs specify the use of thread-local storage data structures according to variant II. Note the use of the plural: there are two versions of the IA-32 ABI. The data structure layout does not differ between the two models. The size of the TCB does not matter for the ABIs. The dynamic thread vector cannot be directly accessed from compiler generated code. Each element of the dtv_t is 4 bytes in size, enough for a pointer and certainly enough for a generation counter.

Since the IA-32 architecture is low on registers the thread register is encoded indirectly through the `%gs` segment register. The only requirement about this register is that the actual thread pointer tp_t can be loaded from the absolute address 0 via the `%gs` register. The following code would load the thread pointer in the `%eax` register:

```
movl %gs:0, %eax
```

To access TLS blocks for modules using the static model the $\mathit{tlsoffset}_m$ offsets have to be known. These values must be **subtracted** from the thread register value. Unlike what happens on IA-64 where the offsets are added. The offsets are computed as follows:

$$\begin{aligned}\mathit{tlsoffset}_1 &= \mathrm{round}(\mathit{tlssize}_1, \mathit{align}_1)\\ \mathit{tlsoffset}_{m+1} &= \mathrm{round}(\mathit{tlsoffset}_m + \mathit{tlssize}_{m+1}, \mathit{align}_{m+1})\end{aligned}$$

for all m in $1 \leq m \leq M$ where M is the total number of modules. These formulas differ slightly from the IA-64 formulas because of the fact that the values have to be subtracted.

The `__tls_get_addr` function also differs slightly from the IA-64 version. The prototype is

```
extern void *__tls_get_addr (tls_index *ti);
```

where the type `tls_index` is defined as

```
typedef struct
  {
     unsigned long int ti_module;
     unsigned long int ti_offset;
  } tls_index;
```

The element names are given only for presentation purposes. They are not available outside the run-time environment. The information passed to the function is the same as for the IA-64 version of this function but only code to pass one parameter must be generated and the values need not be loaded from the GOT by the calling code. Instead this is centralized in the `__tls_get_addr` function. Note that the elements of the structure have the same size as individual elements of the GOT. Therefore such a structure can be defined on the GOT, occupying two GOT entries.

The definition of this function is one of the things which distinguish the two IA-32 ABIs. The ABI defined by Sun Microsystems uses the traditional IA-32 calling convention for this function where the parameter is passed to the function on the stack. The GNU variant of the ABI defines that the parameter is passed to the function in the `%eax` register. To avoid conflicts with the Sun interface the function has a different name (note the **three** leading underscores):

```
extern void *___tls_get_addr (tls_index *ti)
  __attribute__ ((__regparm__ (1)));
```

This declaration uses the notation for the GNU C compiler. The difference for the function itself is not big. But the complexity of the linker operations and the size of the generated code varies greatly in favor of the GNU variant.

For the implementation on GNU systems we can add one more requirement. The address `%gs:0` represents is actually the same as the thread pointer. I.e., the content of the word addressed via `%gs:0` is the address of the very same location. The advantage is potentially big since we can access memory directly via the `%gs` register without loading the thread pointer first. The documentation for the initial and local exec model for x86 below shows the advantages.

3.4.3 SPARC specific

The SPARC ABI is virtually the same as the IA-32 ABI. Both were designed by Sun. The difference between 32-bit and 64-bit SPARC implementations is only the different size of variables containing pointers.

As for IA-32, the structure of the TCB is not specified. The `%g7` register is used as the thread register containing tp_t. Accessing the dynamic thread vector with the thread register's help is implementation defined. Each element of the dtv_t is 4 bytes in size for the 32-bit SPARC and 8 bytes in size for 64-bit SPARC.

The TLS blocks of the modules present at startup time are allocated according to variant II of the data structure layout and the offsets are computed with the same formulas both, the 32- and the 64-bit, code.

$$\begin{aligned} tlsoffset_1 &= \text{round}(tlssize_1, align_1) \\ tlsoffset_{m+1} &= \text{round}(tlsoffset_m + tlssize_{m+1}, align_{m+1}) \end{aligned}$$

for all m in $1 \leq m \leq M$ where M is the total number of modules.

The `__tls_get_addr` function has the same interface as on IA-32. The prototype is

```
extern void *__tls_get_addr (tls_index *ti);
```

where the type `tls_index` is defined as

```
typedef struct
  {
     unsigned long int ti_module;
     unsigned long int ti_offset;
  } tls_index;
```

Here as well the element names are given only for presentation purposes. They are not available outside the run-time environment.

Since the `unsigned long int` type has 4 bytes on 32-bit SPARC and 8 bytes on 64-bit SPARC systems the elements of `tls_index` have for both CPU versions the same size as elements of the GOT and therefore it is here also possible to define object of this type in the GOT data structure.

3.4.4 SH specific

The SH ABI was designed by Kaz Kojima to follow the design of variant I. There is not yet any support for 64-bit SH architectures. The `__tls_get_addr` function has the same interface as on SPARC:

```
extern void *__tls_get_addr (tls_index *ti);
```

where the type `tls_index` is defined as

```
typedef struct
  {
     unsigned long int ti_module;
     unsigned long int ti_offset;
  } tls_index;
```

As usual, the element names are given only for presentation purposes. They are not available outside the run-time environment.

The details for the currently supported SH ABIs differ from the SPARC, IA-32, and IA-64 code because of the architecture of the processor. The processor versions before SH-5 provide only very restricted addressing modes which allow only offsets with up to 12 bits. Since the compiler cannot make any assumptions on the layout and size of functions (and therefore the relative position of symbols) addresses of objects and functions cannot generally be computed at runtime. Instead addresses are stored in variables and the values are computed by the runtime linker at load time. This abolishes the need to define any TLS relocations for instructions. It is only necessary to define relocations for data object. This simplifies the TLS handling significantly since only very few new relocations are needed.

The code sequences to access TLS are fixed. No scheduling is allowed. It is not necessary with the SH implementation today since they do not feature sophisticated out-of-order execution.

3.4.5 Alpha specific

The Alpha ABI is a hybrid between the IA-64 and SPARC models. The thread-local storage data structures follow variant I above. The size of the TCB is 16 bytes where the first 8 bytes contain the pointer to the dynamic thread vector. The other 8 bytes are reserved for the implementation.

The TLS blocks for all modules present at startup time (i.e. those which cannot be unloaded) are created consecutively following the TCB. The $\mathit{tlsoffset}_x$ values are computed as follows:

$$\begin{aligned}\mathit{tlsoffset}_1 &= \mathrm{round}(16, \mathit{align}_1)\\ \mathit{tlsoffset}_{m+1} &= \mathrm{round}(\mathit{tlsoffset}_m + \mathit{tlssize}_m, \mathit{align}_{m+1})\end{aligned}$$

for all m in $1 \leq m \leq M$ where M is the total number of modules.

The `__tls_get_addr` function is defined as for SPARC,

```
extern void *__tls_get_addr (tls_index *ti);
```

The thread pointer is held in the thread's process control block. This value is accessed via the PALcode entry point `PAL_rduniq`.

3.4.6 x86-64 specific

The x86-64 ABI is virtually the same as the IA-32 ABI. The difference is mainly in different size of variables containing pointers and that it only provides one variant which closely matches the IA-32 GNU variant.

Instead of segment register `%gs` it uses the `%fs` segment register. Accessing the dynamic thread vector with the thread register's help is implementation defined. Each element of the dtv_t is 8 bytes in size.

The TLS blocks of the modules present at startup time are allocated according to variant II of the data structure layout and the offsets are computed with the same formulas.

$$\begin{aligned} \textit{tlsoffset}_1 &= \text{round}(\textit{tlssize}_1, \textit{align}_1) \\ \textit{tlsoffset}_{m+1} &= \text{round}(\textit{tlsoffset}_m + \textit{tlssize}_{m+1}, \textit{align}_{m+1}) \end{aligned}$$

for all m in $1 \leq m \leq M$ where M is the total number of modules.

The `__tls_get_addr` function has the same interface as on IA-32. The prototype is

```
extern void *__tls_get_addr (tls_index *ti);
```

where the type `tls_index` is defined as

```
typedef struct
  {
     unsigned long int ti_module;
     unsigned long int ti_offset;
  } tls_index;
```

Here as well the element names are given only for presentation purposes. They are not available outside the run-time environment.

3.4.7 s390 specific

The s390 ABI uses variant II of the thread-local storage data structures. The size of the TCB does not matter for the ABI. The thread pointer is stored in access register `%a0` and needs to get extracted into a general purpose register before it can be used as an address. One way to get the thread pointer from `%a0` to, for example, `%r1` is by use of the `ear` instruction:

```
ear %r1, %a0
```

The TLS blocks of the modules present at startup are allocated according to variant II of the data structure layout and the offsets are computed with the same formulas. The $tlsoffset_i$ values must be subtracted from the thread register value.

$$\begin{aligned} \textit{tlsoffset}_1 &= \text{round}(\textit{tlssize}_1, \textit{align}_1) \\ \textit{tlsoffset}_{m+1} &= \text{round}(\textit{tlsoffset}_m + \textit{tlssize}_{m+1}, \textit{align}_{m+1}) \end{aligned}$$

for all m in $1 \leq m < M$ where M is the total number of modules.

The s390 ABI is defined to use the `__tls_get_offset` function instead of the `__tls_get_addr` function used in other ABIs. The prototype is:

```
unsigned long int __tls_get_offset (unsigned long int offset);
```

The function has a second, hidden parameter. The caller needs to set up the GOT register `%r12` to contain the address of the global offset table of the caller's module. The `offset` parameter, when added to the value of the GOT register, yields the address of a `tls_index` structure located in the caller's global offset table. The type `tls_index` is defined as

```
typedef struct
  {
    unsigned long int ti_module;
    unsigned long int ti_offset;
  } tls_index;
```

The return value of `__tls_get_offset` is an offset to the thread pointer. To get the address of the requested variable the thread pointer needs to be added to the return value. The use of `__tls_get_offset` might seem more complicated than the standard `__tls_get_addr` but for s390 the use of `__tls_get_offset` allows for better code sequences.

3.4.8 s390x specific

The s390x ABI is a close match to the s390 ABI. The thread-local storage data structures follows variant II. The size of the TCB does not matter for the ABI. The thread pointer is stored in the pair of access registers `%a0` and `%a1` with the higher 32 bits of the thread pointer in `%a0` and the lower 32 bits in `%a1`. One way to get the thread pointer into e.g. register `%r1` is to use the following sequence of instructions:

```
ear  %r1,%a0
sllg %r1,%r1,32
ear  %r1,%a1
```

The TLS block allocation of the modules present at startup uses the same formulas for $tlsoffset_m$ as s390 and the s390x ABI uses the same `__tls_get_offset` interface as s390.

4 TLS Access Models

The document so far already mentioned two different ways to access thread-local storage, the dynamic and the static model. These are the basic differentiations of the TLS access models. Different models, falling in one of these two categories, are used to provide as much performance as possible. The ABIs covered in this document define four different access models. The ABIs for other platforms might define additional models.

All models have in common that the dynamic linker at startup-time or when a module gets loaded dynamically has to process all the relocations related to thread-local storage. Processing of none of these relocations can be deferred; just as any other relocation for variables (instead of function calls) they must be processed right away.

When performing a relocation for `STT_TLS` symbol the result is a module ID and a TLS block offset. For relocations or normal symbols the result would be the address of the symbol. The module ID and TLS block offset are then stored in the GOT. The text segment cannot be modified and therefore the code generated by the compiler and linker has instructions which read the values from the GOT.

4.1 General Dynamic TLS Model

The general dynamic TLS model is the most generic. Code compiled with it can be used everywhere and it can access variables defined anywhere else. Compilers will by default generate code with this model and only use a more restrictive model when explicitly told to do so or when it can safely use another model without limiting the generality.

The generated code for this model does not assume that module number nor variable offset is known at link-time (leave alone compile-time). The values for the module ID and the TLS block offset are determined by the dynamic linker at run-time and then passed to the `__tls_get_addr` function in an architecture-specific way. The `__tls_get_addr` function upon return has computed the address of the variable for the current thread.

The size of the code to implement this model and the time needed at run-time for relocation and in the code to compute the address makes it necessary to avoid this model whenever possible. If both the module ID and the TLS block offset or even only the module ID are known better ways are available.

Since in this model the `__tls_get_addr` function is called to calculate the variable address it is possible to defer allocating the TLS block with the techniques described above. If the linker is changing the code to something more efficient this could be a model which does not allow deferred allocation.

In the following sections the code shown is determining a address of a thread-local variable `x`:

```
extern __thread int x;

&x;
```

4.1.1 IA-64 General Dynamic TLS Model

Since the IA-64 version of the `__tls_get_addr` function is expecting the module ID and the TLS block offset as parameters the code sequence for the general dynamic TLS model on IA-64 has to load these two values in the parameter registers `out0` and `out1`. The result will be in the result register `ret0`.

It is important to know that the IA-64 ABI does **not** provide provisions for linker relaxation. Once code is generated for a certain model the linker cannot help even if it could find out that the model is not optimal. It is therefore important that the compiler (sometimes guided by the programmer) generates the right code.

In the code sequences the instructions get assigned addresses of offsets. For IA-64 these only help referring to the instructions easier. The compiler can freely decide to rearrange them.

General Dynamic Model Code Sequence	**Initial Relocation**	**Symbol**
`0x00 mov  loc0=gp`		
`0x06 addl t1=@ltoff(@dtpmod(x)),gp`	`R_IA_64_LTOFF_DTPMOD22`	`x`
`0x0c addl t2=@ltoff(@dtprel(x)),gp`	`R_IA_64_LTOFF_DTPREL22`	`x`
`     ;;`		
`0x10 ld8  out0=[t1]`		
`0x16 ld8  out1=[t2]`		
`0x1c br.call rp=__tls_get_addr`		
`     ;;`		
`0x20 mov  gp=loc0`		
	Outstanding Relocations	
`GOT[m]`	`R_IA_64_DTPMOD64LSB`	`x`
`GOT[n]`	`R_IA_64_DTPREL64LSB`	`x`

The instruction at address `0x06` determines the address of the GOT entry generated for the `@ltoff(@dtpmod(x))` expression. The linker puts the 22-bit offset of the entry from the `gp` register in the instruction and creates a new GOT entry, `GOT[m]` in the example, which gets filled at run-time by the dynamic linker. For this the dynamic linker has to process the `R_IA_64_DTPMOD64LSB` relocation to determine the module ID for the module containing the symbol `x` (on platforms using big-endian the relocation would be `R_IA_64_DTPMOD64MSB`).

Similarly the instruction at address `0x0c` is handled. The assembler handles the `@ltoff(@dtprel(x))` expression by storing `gp`-relative offset of the GOT entry in the instruction and allocating a new GOT entry. The dynamic linker stores at run-time in this GOT entry, `GOT[n]` (where `n` does not have to have any relation to `m`), the offset of the variable `x` in the TLS block of the module the variable was found in. The value is determined by processing the `R_IA_64_DTPREL64LSB` relocation attached to this GOT entry (on big-endian systems it would be `R_IA_64_DTPREL64MSB`).

The remainder of the generated code is straight-forward. The GOT values are loaded with the two `ld8` instructions and stored in the parameter registers for the following call to the function `__tls_get_addr`. We have seen the prototype of this function above and it should be obvious to see that it matches the use in the code here.

Upon return the computed address of the thread-local variable `x` is stored in the register `ret0`.

4.1.2 IA-32 General Dynamic TLS Model

The IA-32 code sequence for the general dynamic model exists in two variants since the function `__tls_get_addr` is called differently as explained above. First the version following Sun's model:

General Dynamic Model Code Sequence	Initial Relocation	Symbol
`0x00 leal  x@dtlndx(%ebx),%edx`	`R_386_TLS_GD_32`	`x`
`0x06 pushl %edx`	`R_386_TLS_GD_PUSH`	`x`
`0x07 call  x@TLSPLT`	`R_386_TLS_GD_CALL`	`x`
`0x0c popl  %edx`	`R_386_TLS_GD_POP`	`x`
`0x0d nop`		
	Outstanding Relocations	
`GOT[n]`	`R_386_TLS_DTPMOD32`	`x`
`GOT[n+1]`	`R_386_TLS_DTPOFF32`	`x`

The `__tls_get_addr` function of the IA-32 ABI only takes one parameter which is the address of the `tls_index` structure containing the information. The `R_386_TLS_GD_32` relocation created for the `x@dtlndx(%ebx)` expression instructs the linker to allocate such a structure in the GOT. The two entries required for the `tls_index` object must of course be consecutive (`GOT[n]` and `GOT[n+1]` in the example code above). These GOT locations get the relocations `R_386_TLS_DTPMOD32` and `R_386_TLS_DTPOFF32` associated with it. The order of the two GOT entries is determined by the order of the appropriate fields in the definition of `tls_info`.

The instruction at address `0x00` only computes the address of the first GOT entry by adding the offset from the beginning of the GOT which is known at link-time to the content of the GOT register `%ebx`. The result is stored in any of the available 32-bit registers. The example code above uses the `%edx` register but the linker is supposed to be able to handle any register used. The address is then passed to the `__tls_get_addr` function on the stack. The `pushl` and `popl` instruction perform this work. They get their own relocations so that the linker can recognize these instructions in case code relaxations are later possible.

The `x@TLSPLT` expression is the call to `__tls_get_addr`. It is not possible to simply write `call __tls_get_addr@plt` since this would provide the assembler no information about the associated symbol (`x` in this case) and so it would not be able to construct the correct relocation. This relocation, once more, is necessary for possible code relaxations.

After the function call the register `%eax` contains the address of the thread-local variable `x`. The `nop` instruction at address `0x0d` is added here to create a code sequence which allows code relaxations to be performed. As we will see later some of the code sequences used for other access models need more space.

The code sequence for the GNU variant is similar but significantly simpler:

General Dynamic Model Code Sequence	Initial Relocation	Symbol
`0x00 leal x@tlsgd(,%ebx,1),%eax`	`R_386_TLS_GD`	`x`
`0x07 call ___tls_get_addr@plt`	`R_386_PLT32`	`___tls_get_addr`
	Outstanding Relocations	
`GOT[n]`	`R_386_TLS_DTPMOD32`	`x`
`GOT[n+1]`	`R_386_TLS_DTPOFF32`	`x`

The different calling convention for `___tls_get_addr` reduces the code sequence by two instructions. The parameter is passed to the function in the `%eax` register. This is what the `leal` instruction at address `0x00` does. To signal that this instruction is for the GNU variant of the access model the syntax `x@tlsgd(%ebx)` is used. This creates the relocation `R_386_TLS_GD` instead of `R_386_TLS_GD_32`. The effect on the GOT is the same. The linker allocates two slots in the GOT and places the offset from the GOT register `%ebx` in the instruction. Note the form of the first operand of `leal` which forces the use of the SIB-form of this instruction, increasing the size of the instruction by one byte and avoiding an additional `nop`.

The call instruction also differs. There is no need for a special relocation and so `___tls_get_addr` is called using the normal syntax for a function call.

4.1.3 SPARC General Dynamic TLS Model

The SPARC general dynamic access model is very similar to the IA-32 one. The `__tls_get_addr` function is called with one parameter which is a pointer to an object of type `tls_index`.

General Dynamic Model Code Sequence	Initial Relocation	Symbol
`0x00 sethi %hi(@dtlndx(x)),%o0`	`R_SPARC_TLS_GD_HI22`	`x`
`0x04 add   %o0,%lo(@dtlndx(x)),%o0`	`R_SPARC_TLS_GD_LO10`	`x`
`0x08 add   %l7,%o0,%o0`	`R_SPARC_TLS_GD_ADD`	`x`
`0x0c call  __tls_get_addr`	`R_SPARC_TLS_GD_CALL`	`x`
	Outstanding Relocations, 32-bit	
`GOT[n]`	`R_SPARC_TLS_DTPMOD32`	`x`
`GOT[n+1]`	`R_SPARC_TLS_DTPOFF32`	`x`
	Outstanding Relocations, 64-bit	
`GOT[n]`	`R_SPARC_TLS_DTPMOD64`	`x`
`GOT[n+1]`	`R_SPARC_TLS_DTPOFF64`	`x`

The expression `@dtlndx(x)` causes the linker to create an object of type `tls_info` in the GOT. Due to SPARC's RISC architecture the offset has to be loaded in two steps in the register `%o0`. The `@dtlndx(x)` expression used with `%hi()` produces a `R_SPARC_TLS_GD_HI22` relocation while the next instruction uses `%lo()` to get the lower 10 bits and this way creates the matching `R_SPARC_TLS_GD_LO10` relocation.

The offset so loaded is that of the first of two consecutive words in the GOT which the linker will add when creating an executable or shared object and which get the relocations `R_SPARC_TLS_DTPMOD64` and `R_SPARC_TLS_DTPOFF64` assigned. These relocations will instruct the dynamic linker to look up the thread-local symbol `x` and store the module ID of the module it is found in into the first word and the offset in the TLS block into the second word.

The `add` instruction at address `0x08` produces the final address. In this example the `%l7` register is expected to contain the GOT pointer. The linker is prepared to deal with any register, though, not only `%l7`. The requirement is only that the GOT register must be the first register in the instruction. To locate the instruction a `R_SPARC_TLS_GD_ADD` relocation is added to the instruction.

The last instruction in the sequence is the call to `__tls_get_addr` which causes a `R_SPARC_TLS_GD_CALL` relocation to be added.

The code sequence must appear in the code as is. It is not possible to move the second `add` instruction in the delay slot of the `call` instruction since the linker would not recognize the instruction sequence.[2]

4.1.4 SH General Dynamic TLS Model

Accessing a TLS variable in the general dynamic model is simply the concatenation of the code to access a global variable and a function call. The global variable contains the offset of the address TLS variable, a value determined by the linker. The called function is `__tls_get_addr`.

General Dynamic Model Code Sequence	**Initial Relocation**	**Symbol**
`0x00 mov.l 1f,r4`		
`0x02 mova  2f,r0`		
`0x04 mov.l 2f,r1`		
`0x06 add   r0,r1`		
`0x08 jsr   @r1`		
`0x0a  add  r12,r4`		
`0x0c bra   3f`		
`0x0e  nop`		
`     .align 2`		
`1:   .long x@tlsgd`	`R_SH_TLS_GD_32`	`x`
`2:   .long __tls_get_addr@plt`		
`3:`		
	Outstanding Relocations	
`GOT[n]`	`R_SH_TLS_DTPMOD32`	`x`
`GOT[n+1]`	`R_SH_TLS_DTPOFF32`	`x`

[2]This is at least what Sun's documentation says and apparently how Sun's linker works. Given the relocations which show exactly what the instructions do this seems not really necessary.

The value stored in the word labeled with `1:` contains the link-time constant offset of the first of two GOT entries which make up the `tls_index` object. The complete address of the object will be computed in the instruction at offset `0x0a`. The second and third instruction compute the address of `__tls_get_addr` with the usual code sequence. In the instruction at offset `0x08` the function is then called and it returns in `r0` the result. Note that the `add` instruction at offset `0x0a` is executed in the branch delay slot. After `__tls_get_addr` returns all that is necessary is to skip over the data.

It is worth mentioning that this code is fairly expensive. Each and every access to a TLS variable in the general dynamic model requires four words of data and two additional instructions to skip over the data placed in the middle of the text segment.

4.1.5 Alpha General Dynamic TLS Model

The Alpha general dynamic access model is similar to that for IA-32. The `__tls_get_addr` function is called with one parameter which is a pointer to an object of type `tls_index`.

General Dynamic Model Code Sequence	Initial Relocation	Symbol
`0x00 lda  $16,x($gp)  !tlsgd!1`	`R_ALPHA_TLSGD`	`x`
`0x04 ldq  $27,__tls_get_addr($gp)!literal!1`	`R_ALPHA_LITERAL`	`__tls_get_addr`
`0x08 jsr  $26,($27),0 !lituse_tlsgd!1`	`R_ALPHA_LITUSE`	`4`
`0x0c ldah $29,0($26)  !gpdisp!2`	`R_ALPHA_GPDISP`	`4`
`0x10 lda  $29,0($29)  !gpdisp!2`		
	Outstanding Relocations	
`GOT[n]`	`R_ALPHA_DTPMOD64`	`x`
`GOT[n+1]`	`R_ALPHA_DTPREL64`	`x`

The relocation specifier `!tlsgd` causes the linker to create an object of type `tls_info` in the GOT. The address of this object is loaded into the first argument register `$16` with the `lda` instruction. The rest of the sequence is the standard call sequence for a function, except that `!lituse_tlsgd` is used instead of `!lituse_jsr`. The reason for this will become apparent when relaxation is discussed.

4.1.6 x86-64 General Dynamic TLS Model

The x86-64 general dynamic access model is very similar to the IA-32 GNU variant. The `__tls_get_addr` function is called with one parameter which is a pointer to an object of type `tls_index`.

General Dynamic Model Code Sequence	Initial Relocation	Symbol
`0x00 .byte 0x66`		
`0x01 leaq  x@tlsgd(%rip),%rdi`	`R_X86_64_TLSGD`	`x`
`0x08 .word 0x6666`		
`0x0a rex64`		
`0x0b call  __tls_get_addr@plt`	`R_X86_64_PLT32`	`__tls_get_addr`

	Outstanding Relocations	
`GOT[n]`	`R_X86_64_DTPMOD64`	`x`
`GOT[n+1]`	`R_X86_64_DTPOFF64`	`x`

The `__tls_get_addr` function of the x86-64 ABI only takes one parameter which is the address of the `tls_index` structure containing the information. The `R_X86_64_TLSGD` relocation created for the `x@tlsgd(%rip)` expression instructs the linker to allocate such a structure in the GOT. The two entries required for the `tls_index` object must of course be consecutive (`GOT[n]` and `GOT[n+1]` in the example code above). These GOT locations get the relocations `R_X86_64_DTPMOD64` and `R_X86_64_DTPOFF64` associated with it.

The instruction at address `0x00` only computes the address of the first GOT entry by adding the PC relative address of the beginning of the GOT which is known at link-time to the current instruction pointer. The result is passed via the `%rdi` register to the `__tls_get_addr` function. Note the instruction must be preceeded by a `data16` prefix and immediately followed by the `call` instruction at offset `0x08`. The `call` instruction has to be preceeded by two `data16` prefixes and one `rex64` prefix to increase the total size of the whole sequence to 16 bytes. Prefixes and not no-op instructions are used since the former have no negative impact in the code.

4.1.7 s390 General Dynamic TLS Model

For the s390 general dynamic access model the compiler has to set up the GOT register `%r12` before it can call `__tls_get_offset`. The `__tls_get_offset` function gets one parameter which is a GOT offset to an object of type `tls_index`. The return value of the function call has to be added to the thread pointer to get the address of the requested variable.

General Dynamic Model Code Sequence	Initial Relocation Symbol	
`     l   %r6,.L1-.L0(%r13)`		
`     ear %r7,%a0`		
`     l   %r2,.L2-.L0(%r13)`		
`     bas %r14,0(%r6,%r13)`	`R_390_TLS_GDCALL`	`x`
`     la  %r8,0(%r2,%r7) # %r8 = &x`		
`     ...`		
`.L0: # literal pool, address in %r13`		
`.L1: .long __tls_get_offset@plt-.L0`		
`.L2: .long x@tlsgd`	`R_390_TLS_GD32`	`x`
	Outstanding Relocations	
`GOT[n]`	`R_390_TLS_DTPMOD`	`x`
`GOT[n+1]`	`R_390_TLS_DTPOFF`	`x`

The `R_390_TLS_GD32` relocation created for the literal pool entry `x@tlsgd` instructs the linker to allocate a `tls_index` structure in the GOT, occupying two consecutive GOT entries. These two GOT entries have the relocations `R_390_TLS_DTPMOD` and `R_390_TLS_DTPOFF` associated with them.

The `R_390_TLS_GDCALL` relocation tags the instruction to call `__tls_get_offset`. This instructions is subject to TLS model optimization. The tag is necessary because the linker needs to known the location of the call to be able to replace it with an instruction of a different TLS model. How the instruction tag is specified in the assembler syntax is up to the assembler implementation.

The instruction sequence is divided into four parts. The first part extracts the thread pointer from `%a0` and loads the branch offset to `__tls_get_offset`. The first part can be reused for other TLS accesses. A second TLS access doesn't have to repeat these two instruction, but can use `%r6` and `%r7` if these registers have not been clobbered between the two TLS accesses. The second part is the core of the TLS access. For every variable that is accessed by the general dynamic access model these two instruction have to be present. The first loads the GOT offset to the variables `tls_index` structure from the literal pool and the second calls `__tls_get_offset`. The third part uses the extracted thread pointer in `%r7` and the offset in `%r2` returned by the call to `__tls_get_offset` to perform an operation on the variable. In the example the address of `x` is loaded to register `%r8`. The compiler can choose any other suitable instruction to access `x`, for example a "`l %r8,0(%r2,%r7)`" would load the content of `x` to `%r8`. That leaves room for optimizations in the compiler. The fourth part is the literal pool that needs to have an entry for the `x@tlsgd` offset.

All the instruction in the general dynamic access model for s390 can be scheduled freely by the compiler as long as the obvious data dependencies are fulfilled and the registers `%r0` - `%r5` do not contain any information that is still needed after the `bas` instruction (they get clobbered by the function call). Registers `%r6`, `%r7` and `%r8` are not fixed, they can be replaced by any other suitable register.

4.1.8 s390x General Dynamic TLS Model

The general dynamic access model for s390x is more or less a copy of the general dynamic model for s390. The main differences are the more complicated code for the thread pointer extraction, the use of the `brasl` instruction instead of the `bas` and the fact the s390x uses 64 bit offsets.

General Dynamic Model Code Sequence	Initial Relocation Symbol
`ear   %r7,%a0`	
`sllg  %r7,32`	
`ear   %r7,%a1`	
`lg    %r2,.L1-.L0(%r13)`	
`brasl %r14,__tls_get_offset@plt`	`R_390_TLS_GDCALL    x`

```
      la    %r8,0(%r2,%r7) # %r8 = &x   |
      ...                               |
.L0: # literal pool, address in %r13   |
.L2: .quad x@tlsgd                      | R_390_TLS_GD64     x
----------------------------------------+---------------------------
                                        | Outstanding Relocations
GOT[n]                                  | R_390_TLS_DTPMOD   x
GOT[n+1]                                | R_390_TLS_DTPOFF   x
```

The relocations `R_390_TLS_GD64`, `R_390_TLS_DTPMOD` and `R_390_TLS_DTPOFF` do the same as their s390 counterparts, only the bit size of the relocation target is 64 bit instead of 32 bit.

4.2 Local Dynamic TLS Model

The local dynamic TLS model is an optimization of the general dynamic TLS model. The compiler can generate code following this model if it can recognize that the thread-local variable is defined in the same object it is referenced in. This includes, for instance, thread-local variables with file scope or variables which are defined to be protected or hidden (see the Generic ELF ABI specification for more information on this). We refer to these kind of variables here as protected.

Just as a reminder, a thread-local variable is defined by the module ID and the offset in the TLS block of that module. In the case of variables which are known to be found in the same object as the references the offsets are known at link-time. The module ID is not known (unless it is the main application in which case more optimizations can be performed). It is therefore still necessary to call `__tls_get_addr` to get the module ID and eventually allocate the TLS block. If the parameters for `__tls_get_addr` would make the function compute the start address of the TLS block by passing zero as the offset it is then possible to reuse this value many times to access many variables by adding the offset of the protected thread-local variable to the start address of the TLS block. The compiler can easily and efficiently generate such code.

But one must keep in mind that it is normally not really an advantage to use the local dynamic model if only one protected thread-local variable is used this way[3]. It would mean a call to `__tls_get_addr` as for the general dynamic model plus an additional addition to compute the address. But the equation changes if more than one variable is treated this way. We still have only one function call and every variable adds an addition. Because the difference between the general and the local dynamic model is not just replacing some instructions with a few others but instead generating quite different code, the optimization from the general to the local dynamic model cannot be performed by the linker. The compiler has to do it, perhaps with help from the programmer.

In the architecture specific description the examples implement something equivalent to this piece of code:

[3]For IA-64 it can be of advantage.

```
static __thread int x1;
static __thread int x2;

&x1;
&x2;
```

4.2.1 IA-64 Local Dynamic TLS Model

The instruction set of the IA-64 makes it possible that the code sequence to determine the address of one variable with the local dynamic model is shorter than the general dynamic model code sequence. In addition the variable offset does not have to be computed by the dynamic linker and the GOT needs one less element.

Local Dynamic Model Code Sequence	Initial Relocation	Symbol
`0x00 mov  loc0=gp`		
`0x06 addl t1=@ltoff(@dtpmod(x)),gp`	`R_IA_64_LTOFF_DTPMOD22`	`x`
`0x0c addl out1=@dtprel(x),r0`	`R_IA_64_DTPREL22`	`x`
`     ;;`		
`0x10 ld8  out0=[t1]`		
`0x16 br.call rp=__tls_get_addr`		
`     ;;`		
`0x20 mov  gp=loc0`		
	Outstanding Relocations	
`GOT[n]`	`R_IA_64_DTPMOD64LSB`	`x`

The difference to the general dynamic model is that it is not necessary to find the offset of the variable by adding the `@ltoff(@dtprel(x))` value to `gp` and then load from this address. Instead the `addl` instruction at address `0x0c` is used to compute the offset directly (this is how loading an immediate value on IA-64 works). This all means that one less `ld8` instruction is needed in the second bundle and the compiler could fill the slot with something else.

This code sequence has one limitation, though. The offset in the TLS block has only 21 bits. If the amount of thread-local data exceeds 2^{21} bytes (2 MiBi) different code has to be used. Larger offsets must be loaded using the long move instruction which allows a full 64-bit offset to be loaded. In addition, the compiler could optimize the `addl` instruction further if it would be known that the thread-local data requirements don't exceed 2^{13} bytes (8 KiBi). The relocations used in these cases would be `R_IA_64_DTPREL64I` and `R_IA_64_DTPREL14` respectively. Whatever the compiler chooses, there is normally no possibility for the linker to determine the best or necessary instruction so the selection should be up to the user with the help of compiler switches. The code sequence from the example above is a good compromise and useful as the default.

In case a function must access more than one protected thread-local variable the savings can be even larger. In this case the `__tls_get_addr` call is not used to compute the address of any variable but instead only to compute the address of the beginning of the TLS block.

Local Dynamic Model Code Sequence, II	Initial Relocation	Symbol
`0x00 mov  loc0=gp`		
`0x06 addl t1=@ltoff(@dtpmod(x1)),gp`	R_IA_64_LTOFF_DTPMOD22	x1
`0x0c mov  out1=r0`		
`     ;;`		
`0x10 ld8  out0=[t1]`		
`0x16 br.callrp=__tls_get_addr`		
`     ;;`		
`0x20 mov  gp=loc0`		
`0x26 mov  r2=ret0`		
`     ;;`		
`0x30 addl loc1=@dtprel(x1),r2`	R_IA_64_DTPREL22	x1
`0x36 addl loc2=@dtprel(x2),r2`	R_IA_64_DTPREL22	x2
	Outstanding Relocations	
`GOT[n]`	R_IA_64_DTPMOD64LSB	x

The first part of the code is very similar to the previous code where only one variable was used. The only difference is that explicitly zero is passed to `__tls_get_addr` as the second parameter. This computes the beginning of the TLS block for the module `x1` is found in, i.e., the module this code is in as well.

To complete the computations additional code is needed and it starts with saving the return value of the function call in a place where it can later be used (the register `r2`). Finally we see the actual code to compute the variable addresses. It is very simple since we only have to add the offset of the variable to the base address of the TLS block. The offset is an immediate value known at link-time replaced in the code with the `R_IA_64_DTPREL22` relocation. This relocation is, just as in the code above, a compromise between size and flexibility. Here as well the compiler could use the short add instruction or the long move instruction.

4.2.2 IA-32 Local Dynamic TLS Model

The code sequence for the local dynamic model is not providing any advantage over the general dynamic model unless more than one variable is used. It is easy to see why. The code to call `__tls_get_addr` does not change at all since it only computes the address of the GOT entry. The GOT entry must consists of two words even though the `ti_offset` word is known at link-time. In case more than one variable is needed there is an advantage in using this model. The following is the code sequence for Sun's variant.

Local Dynamic Model Code Sequence	Initial Relocation	Symbol
`0x00 leal  x1@tmdnx(%ebx),%edx`	R_386_TLS_LDM_32	x1
`0x06 pushl %edx`	R_386_TLS_LDM_PUSH	x1
`0x07 call  x1@TLSPLT`	R_386_TLS_LDM_CALL	x1
`0x0c popl  %edx`	R_386_TLS_LDM_POP	x1
`     ...`		
`0x10 movl  $x1@dtpoff,%edx`	R_386_TLS_LDO_32	x1
`0x15 addl  %eax,%edx`		

`...`		
`0x20 movl $x2@dtpoff,%edx`	`R_386_TLS_LDO_32`	`x2`
`0x25 addl %eax,%edx`		
	Outstanding Relocations	
`GOT[n]`	`R_386_TLS_DTPMOD32`	`x1`

The `x1@tmdnx(%ebx)` expression in the first instruction instructs the assembler to generate a `R_386_TLS_LDM_32`. This in turn will tell the linker to create a special `tls_index` object on the GOT where the `ti_offset` element is zero. This is why in the code above there is only one outstanding relocation for the GOT. The `ti_module` element will be filled with the module ID of the module the code is in when it processes the `R_386_TLS_DTPMOD32` relocation.

When the call to `__tls_get_addr` call returns the `%eax` register contains the address of the TLS block of the module the code is in for the current thread. All that is needed is to complete the address computation by adding the variable offsets. The instructions at address `0x10` and `0x15` compute the address of the variable `x1` by adding the offset to the `%eax` register content. For this the expression `$x1@dtpoff` is used which generates a relocation of type `R_386_TLS_LDO_32`. This relocation reference the variable `x1` and its offset can be computed by the linker and filled in the instruction.

Using a second variable requires only the repetition of the addition which is less work than the function call and although two variables are used only one `tls_index` element is created in the GOT.

The advantages are even more obvious in the code sequence for the GNU variant.

Local Dynamic Model Code Sequence	**Initial Relocation**	**Symbol**
`0x00 leal x1@tlsldm(%ebx),%eax`	`R_386_TLS_LDM`	`x1`
`0x06 call ___tls_get_addr@plt`	`R_386_PLT32`	`___tls_get_addr`
`     ...`		
`0x10 leal x1@dtpoff(%eax),%edx`	`R_386_TLS_LDO_32`	`x1`
`     ...`		
`0x20 leal x2@dtpoff(%eax),%edx`	`R_386_TLS_LDO_32`	`x2`
	Outstanding Relocations	
`GOT[n]`	`R_386_TLS_DTPMOD32`	`x1`

The computation of the base address in the TLS follows the Sun variant, along with the improvements due to the calling conventions of `___tls_get_addr`. The GOT contains one special `tls_index` entry with the `ti_offset` element being zero. The only differences are that the expression `x1@tlsldm(%ebx)` is used for the address of the GOT entry. The expression is handled just like `x1@tmdnx(%ebx)` except that the relocation which is created for the instruction is `R_386_TLS_LDM` instead of `R_386_TLS_LDM_32`.

But the calling convention is not the only advantage. The instructions to compute the final addresses are optimized as well. Using the power of the `leal` instruction the two instructions needed in Sun's variant can be folded in one. The relocation for the instruction remains the same. But this is not all. If instead of computing the address of the variable the value of it has to be loaded one simply uses

```
movl x1@dtpoff(%eax),%edx
```

This instruction would get the same relocation as the original leal instruction. Storing something in such a variable works exactly the same way.

As long as the base address of the TLS block is kept around in a register loading, storing, or computing the address of a protected thread-local variable is a matter of one instruction.

4.2.3 SPARC Local Dynamic TLS Model

For SPARC as for IA-32 the local dynamic model does not provide any advantage when only one variable is used. The disadvantage is even bigger for SPARC due to the nature of the RISC instruction set. If more than one variable is used the generated code could look like this:

Local Dynamic Model Code Sequence	**Initial Relocation**	**Symbol**
`0x00 sethi %hi(@tmdnx(x1)),%o0`	`R_SPARC_TLS_LDM_HI22`	`x1`
`0x04 add   %o0,%lo(@tmndx(x1)),%o0`	`R_SPARC_TLS_LDM_LO10`	`x1`
`0x08 add   %l7,%o0,%o0`	`R_SPARC_TLS_LDM_ADD`	`x1`
`0x0c call  __tls_get_addr`	`R_SPARC_TLS_LDM_CALL`	`x1`
`     ...`		
`0x10 sethi %hix(@dtpoff(x1)),%l1`	`R_SPARC_TLS_LDO_HIX22`	`x1`
`0x14 xor   %l1,%lox(@dtpoff(x1)),%l1`	`R_SPARC_TLS_LDO_LOX22`	`x1`
`0x18 add   %o0,%l1,%l1`	`R_SPARC_TLS_LDO_ADD`	`x1`
`     ...`		
`0x20 sethi %hix(@dtpoff(x2)),%l2`	`R_SPARC_TLS_LDO_HIX22`	`x2`
`0x24 xor   %l2,%lox(@dtpoff(x2)),%l2`	`R_SPARC_TLS_LDO_LOX22`	`x2`
`0x28 add   %o0,%l2,%l2`	`R_SPARC_TLS_LDO_ADD`	`x2`
	Outstanding Relocations, 32-bit	
`GOT[n]`	`R_SPARC_TLS_DTPMOD32`	`x1`
	Outstanding Relocations, 64-bit	
`GOT[n]`	`R_SPARC_TLS_DTPMOD64`	`x1`

The first four instructions are basically equivalent to the code sequence used for the general dynamic model. But instead of using `@dtlndx(x)` to generate a `tls_index` entry for symbol `x` this code uses `tmndx(x1)` which creates a special kind of index which refers to the current module (which contains `x1`) with an offset zero. The linker will create only one relocation for the object, depending on the platform either `R_SPARC_TLS_DTPMOD32` or `R_SPARC_TLS_DTPMOD64`. The `DTPREL` relocation is not necessary.

The reason for this is that the offsets are loaded separately. The `@dtpoff(x1)` expression is used to access the offset of the symbol `x1`. Using the two instructions at address `0x10` and `0x14` the complete offset is loaded and added to the result of the `__tls_get_addr` call in `%o0` to produce the result in `%l1`. The `@dtpoff(x1)` expressions creates the relocations `R_SPARC_TLS_LDO_HIX22` and `R_SPARC_TLS_LDO_LOX22` for the `%hix()` and `%lox()` part respectively. The `add` instruction is marked with a `R_SPARC_TLS_LDO_ADD` relocation so that the linker can recognize it.

The benefit of using the local dynamic model is that for every additional variable only three new instructions have to be added and no additional GOT entries or run-time relocations. Altogether, it **might** be even preferable to use this model even for one variable if the run-time overhead of processing the run-time relocations should be avoided.

4.2.4 SH Local Dynamic TLS Model

As for the other architectures the code generated for the local dynamic model in SH differs from the general dynamic model in that for the first local symbol which is looked up additional efforts are necessary. The code sequence for the second and all later lookups is much cheaper which is especially true for SH.

Local Dynamic Model Code Sequence	**Initial Relocation**	**Symbol**
`0x00 mov.l 1f,r4`		
`0x02 mova  2f,r0`		
`0x04 mov.l 2f,r1`		
`0x06 add   r0,r1`		
`0x08 jsr   @r1`		
`0x0a  add  r12,r4`		
`0x0c bra   3f`		
`0x0e  nop`		
`     .align 2`		
`1:   .long x1@tlsgd`	`R_SH_TLS_LD_32`	`x1`
`2:   .long __tls_get_addr@plt`		
`3:   ...`		
`     mov.l .Lp,r1`		
`     mov.l r0,r1`		
`     ...`		
`     mov.l .Lq,r1`		
`     mov.l r0,r1`		
`     ...`		
`.Lp: .long x1@dtpoff`	`R_SH_TLS_LDO_32`	`x1`
`.Lp: .long x2@dtpoff`	`R_SH_TLS_LDO_32`	`x2`
	Outstanding Relocations	
`GOT[n]`	`R_SH_TLS_DTPMOD32`	`x1`

The first seven instruction are equivalent to those in the generic dynamic model. Only this time the symbol looked up is special as it has the offset zero in the module's TLS data segment. This is identical to what is done on SPARC and IA-32. The difference to the generic dynamic code is that only one of the two GOT slots needed has a relocation attached. The `ti_offset` field is always zero.

Once these preliminaries are over the code to determine the address of the local variables is simply. It consists of loading the linktime-constant offset of the variable in the TLS segment and adding to it the earlier found address of the beginning of the TLS segment for module and the current thread.

Compared with the generic dynamic model code sequence a lookup of two variables saves three instructions, one GOT entry, and one function call. For three TLS variable lookups the benefit would be eight instructions, one data word, two GOT entries, and two function calls. It is easy to see that choosing the local dynamic model pays off whenever more than one variable is in play.

It is worth noting that in this code sequence the allocation of the memory for the offsets for the variables, marked by the labels `.Lp` and `.Lq`, can be delayed and eventually combined with other data (as in the example code above). The `mov.l` and `add` instructions do not have to be touched again after they have been created. Optimizations of the local dynamic model to the local exec model do not touch these instructions. Therefore they can be moved around freely by the compiler, they need not have a fixed relative position to the data.

4.2.5 Alpha Local Dynamic TLS Model

For Alpha as for IA-32 the local dynamic model does not provide any advantage when only one variable is used. If more than one variable is used the generated code could look like this:

Local Dynamic Model Code Sequence	**Initial Relocation**	**Symbol**
`0x00 lda  $16,x($gp)  !tlsldm!1`	R_ALPHA_TLSLDM	x
`0x04 ldq  $27,__tls_get_addr($gp)!literal!1`	R_ALPHA_LITERAL	__tls_get_addr
`0x08 jsr  $26,($27),0 !lituse_tlsldm!1`	R_ALPHA_LITUSE	5
`0x0c ldah $29,0($26)  !gpdisp!2`	R_ALPHA_GPDISP	4
`0x10 lda  $29,0($29)  !gpdisp!2`		
`     ...`		
`0x20 lda  $1,x1($0)   !dtprel`	R_ALPHA_DTPREL16	x1
`     ...`		
`0x30 ldah $1,x2($0)   !dtprelhi`	R_ALPHA_DTPRELHI	x2
`0x34 lda  $1,x2($1)   !dtprello`	R_ALPHA_DTPRELLO	x2
`     ...`		
`0x40 ldq  $1,x3($gp)  !gotdtprel`	R_ALPHA_GOTDTPREL	x3
`0x44 addq $0,$1,$1`		
	Outstanding Relocations	
`GOT[n]`	R_ALPHA_DTPMOD64	x

The instructions between `0x00` and `0x14` are basically the same as the sequence used for the general dynamic model. The difference is that `!tlsldm` is used instead of `!tlsgd`, which creates a `tls_index` entry for the current object with a zero offset.

The offset is added later with one of the `dtprel` relocations. For this we have three choices of code generation options depending on the expected size of the TLS data segment. The sequence at `0x20` is good for a 15 bit positive displacement (32 KiB); the sequence at `0x30` is good for a 31 bit positive displacement (2 GiB); and the final sequence at `0x40` is good for a 64 bit displacement.

4.2.6 x86-64 Local Dynamic TLS Model

Similarly to IA-32 and SPARC this access model has no advantage over global dynamic model if there is just one local variable accessed this way.

Local Dynamic Model Code Sequence	Initial Relocation	Symbol
`0x00 leaq x1@tlsld(%rip),%rdi`	`R_X86_64_TLSLD`	`x1`
`0x07 call __tls_get_addr@plt`	`R_X86_64_PLT32`	`__tls_get_addr`
`...`		
`0x10 leaq x1@dtpoff(%rax),%rcx`	`R_X86_64_DTPOFF32`	`x1`
`...`		
`0x20 leaq x2@dtpoff(%rax),%r9`	`R_X86_64_DTPOFF32`	`x2`
	Outstanding Relocations	
`GOT[n]`	`R_X86_64_DTPMOD64`	`x1`

The first two instructions are basically equivalent to the code sequence used for the general dynamic model, although lack any padding. The two instructions must be consecutive. Instead of using `x1@tlsgd(%rip)` to generate a `tls_index` entry for symbol `x1` this code uses `x1@tlsld(%rip)` which creates a special kind of index which refers to the current module (which contains `x1`) with an offset zero. The linker will create only one relocation for the object, `R_X86_64_DTPMOD64`. The `R_X86_64_DTPOFF64` relocation is not necessary.

The reason for this is that the offsets are loaded separately. The `x1@dtpoff` expression is used to access the offset of the symbol `x1`. Using the instruction at address `0x10` the complete offset is loaded and added to the result of the `__tls_get_addr` call in `%rax` to produce the result in `%rcx`. The `x1@dtpoff` expression creates the `R_X86_64_DTPOFF32` relocation. If instead of computing the address of the variable the value of it has to be loaded one simply uses

```
  movq x1@dtpoff(%rax),%r11
```

This instruction would get the same relocation as the original leaq instruction. Storing something in such a variable works exactly the same way.

As long as the base address of the TLS block is kept around in a register loading, storing, or computing the address of a protected thread-local variable is a matter of one instruction.

The benefit of using the local dynamic model is that for every additional variable only three new instructions have to be added and no additional GOT entries or run-time relocations. Altogether, it **might** be even preferable to use this model even for one variable if the run-time overhead of processing the run-time relocations can be avoided.

4.2.7 s390 Local Dynamic TLS Model

The code sequence of the local dynamic TLS model for s390 does not provide any advantage over the general dynamic model if only a single variable is accessed. It is even slightly worse because an additional literal pool entry is needed (`x@tlsldm` and `x@dtpoff` instead of just `x@tlsgd`) that has to get loaded and added to the return value of the `__tls_get_offset` function call. The local dynamic model is much better than the global dynamic model if more than a single local variable is accessed because for every additional variable only a simple literal pool load is needed instead of a full blown function call.

Local Dynamic Model Code Sequence	Initial Relocation Symbol	
`     l   %r6,.L1-.L0(%r13)`		
`     ear %r7,%a0`		
`     l   %r2,.L2-.L0(%r13)`		
`     bas %r14,0(%r6,%r13)`	`R_390_TLS_LDCALL`	`x1`
`     la  %r8,0(%r2,%r7)`		
`     l   %r9,.L3-.L0(%r13)`		
`     la  %r10,0(%r10,%r8) # %r10 = &x1`		
`     l   %r9,.L4-.L0(%r13)`		
`     la  %r10,0(%r10,%r8) # %r10 = &x2`		
`     ...`		
`.L0: # literal pool, address in %r13`		
`.L1: .long __tls_get_offset@plt-.L0`		
`.L2: .long x1@tlsldm`	`R_390_TLS_LDM32`	`x1`
`.L3: .long x1@dtpoff`	`R_390_TLS_LDO32`	`x1`
`.L4: .long x2@dtpoff`	`R_390_TLS_LDO32`	`x2`
	Outstanding Relocations	
`GOT[n]`	`R_390_TLS_DTPMOD`	`x1`

As for the IA-32 local dynamic TLS model semantic the `x1@tlsldm` expression in the literal pool instructs the assembler to emit a `R_390_TLS_LDM32` relocations. The linker will create a special `tls_index` object on the GOT for it with the `ti_offset` element set to zero. The `ti_module` element will be filled with the module ID of the module the code is in when it processes the `R_390_TLS_LDM32` relocation. The literal pool entries `x1@dtpoff` and `x2@dtpoff` are translated by the assembler into `R_390_TLS_LDO32` relocations. The linker will calculate the offsets for `x1` and `x2` in the TLS block for the module and will write them to the literal pool.

The instruction sequence is divided into four parts. The first part is analog to the first part of the general dynamic model. The second part calls `__tls_get_offset` with the GOT offset to the special `tls_index` object created through the `x@tlsldm` entry in the literal pool. The GOT register `%r12` has to be set up before the call. After the third instruction in the second part of the code sequence `%r8` contains the address of the thread local memory for the module the code is in. Part three of the code sequence shows how the addresses of the thread local variable `x1` and `x2` are calculated. Part four shows the literal pool entries needed by the code sequence.

All the instruction of the local dynamic code sequence can be scheduled freely by the compiler as long as the obvious data dependencies are fulfilled and the function call semantic of the `bas` instruction is taken into account.

4.2.8 s390x Local Dynamic TLS Model

The local dynamic access model for s390x is similar to the s390 version. The same differences as between the two general dynamic models for s390 vs. s390x are present. The extraction of the thread pointer requires three instruction instead of one, the branch to `__tls_get_offset` is done with the `brasl` instruction and the offsets have 64 bit instead of 32 bit.

Local Dynamic Model Code Sequence	**Initial Relocation**	**Symbol**
`    ear   %r7,%a0`		
`    sllg  %r7,%r7,32`		
`    ear   %r7,%a1`		
`    lg    %r2,.L1-.L0(%r13)`		
`    brasl %r14,__tls_get_offset@plt`	`R_390_TLS_LDCALL`	`x1`
`    la    %r8,0(%r2,%r7)`		
`    lg    %r9,.L2-.L0(%r13)`		
`    la    %r10,0(%r9,%r8) # %r10 = &x1`		
`    lg    %r9,.L3-.L0(%r13)`		
`    la    %r10,0(%r9,%r8) # %r10 = &x2`		
`    ...`		
`.L0: # literal pool, address in %r13`		
`.L1: .quad x1@tlsldm`	`R_390_TLS_LDM64`	`x1`
`.L2: .quad x1@dtpoff`	`R_390_TLS_LDO64`	`x1`
`.L3: .quad x2@dtpoff`	`R_390_TLS_LDO64`	`x2`
	Outstanding Relocations	
`GOT[n]`	`R_390_TLS_DTPMOD`	`x1`

4.3 Initial Exec TLS Model

A more restrictive optimization is usable if the variables accessed are known to be in one of the modules available and program start and if the programmer selects to use the static access model. The last condition means that the generated code will not use the `__tls_get_addr` function which means that deferred allocation of memory for the TLS blocks accessed this way is not possible. It would still be possible to defer allocation for dynamically loaded modules.

The idea behind the optimization is that after the dynamic linker loaded all modules referenced directly and indirectly by the executable (and some more like those named by `LD_PRELOAD`) each variable in the TLS block of any of those modules has a fixed offset from the TCB since all the memory for the initially loaded modules is required to be allocated consecutively. The offsets are computed using the architecture-specific formulas for $\mathit{tlsoffset}_m$ described in section 3.4 (where m is the module ID of the module the variable is found in) to which the offset of the variable in the TLS block is added.

The consequence of this optimization is that for each variable there would be a run-time relocation for a GOT entry which instructs the dynamic linker to compute the offset from the TCB. There is no need to compute the module ID. Therefore, coming from the general dynamic model, the number of run-time relocations is cut by half.

The code sequences in the following discussion implement a simple access to a variable `x`:

```
extern __thread int x;

&x;
```

4.3.1 IA-64 Initial Exec TLS Model

The initial exec model requires the code sequence to get the offset relative to the TCB from the GOT location the dynamic linker put it in and add this value to the thread pointer. Very short and simple.

Initial Exec Model Code Sequence	**Initial Relocation**	**Symbol**
`0x00 addl t1=@ltoff(@tprel(x)),gp`	`R_IA_64_LTOFF_TPREL22`	`x`
`     ;;`		
`0x10 ld8  t2=[t1]`		
`     ;;`		
`0x20 add  loc0=t2,tp`		
	Outstanding Relocations	
`GOT[n]`	`R_IA_64_TPREL64LSB`	`x`

The `@ltoff(@tprel(x))` expression instructs the linker to create a `R_IA_64_LTOFF_TPREL22` relocation which in turn requests the linker to create a GOT entry with a `R_IA_64_TPREL64LSB` relocation associated. This relocation is processed at program startup time by the dynamic linker to produce an offset relative to the TCB block (pointed to by the `tp` register) of the desired variable. The offset value only has to be loaded with the `ld8` instruction at address `0x10` and then added to the value of the `tp` register to get the final address in the `loc0` register.

The instructions can be freely mixed with other to enhance policy. Especially the handling of the `tp` register handling can be optimized.

4.3.2 IA-32 Initial Exec TLS Model

The IA-32 code for the initial exec model is very simple and fast. The only problem is locating the TCB block. The mechanism for this used by the platforms supported so far is to use the `%gs` segment register. Accessing memory at offset 0 with this segment register enables loading the TCB address. As always we handle Sun's version first.

Initial Exec Model Code Sequence	Initial Relocation	Symbol

`0x00 movl x@tpoff(%ebx),%edx`
`0x06 movl %gs:0,%eax`
`0x0c subl %edx,%eax`
	`R_386_TLS_IE_32`	`x`

	Outstanding Relocations	
`GOT[n]`	`R_386_TLS_TPOFF32`	`x`

The assembler generates for the `x@tpoff(%ebx)` expressions a `R_386_TLS_IE_32` relocation for the symbol `x` which requests the linker to generate a GOT entry with a `R_386_TLS_TPOFF32` relocation. The offset of the GOT entry is then used in the instruction. The `R_386_TLS_TPOFF32` relocation is processed at program startup time by the dynamic linker by looking up the symbol `x` in the modules loaded at that point. The offset is written in the GOT entry and later loaded by the instruction at address `0x00` in the `%edx` register.

The `movl` instruction at address `0x06` loads the thread pointer for the current thread in the `%eax` register. This step eventually has to be adjusted to the method the platform is using to access the thread pointer.

Finally, the `subl` instruction computes the final address. Note that it is necessary to subtract the offset from the thread pointer. In variant II of the thread-local storage data structure which IA-32 uses the TLS blocks are located before the TCB.

This code sequence requires only three instructions and occupies 14 bytes just like the general dynamic model code sequence. It can be done better as the GNU variants shows. There are two different GNU variants, one for position independent code which uses GOT pointer and one for code without GOT pointer. The position independent variant:

Initial Exec Model Code Sequence, II	Initial Relocation	Symbol
`0x00 movl %gs:0,%eax`		
`0x06 addl x@gotntpoff(%ebx),%eax`	`R_386_TLS_GOTIE`	`x`
	Outstanding Relocations	
`GOT[n]`	`R_386_TLS_TPOFF`	`x`

This code sequence does basically the same except that the GOT value is added, not subtracted, it combines the loading from the GOT and the arithmetic in one instruction. The variant without GOT pointer is:

Initial Exec Model Code Sequence, III	**Initial Relocation**	**Symbol**
`0x00 movl %gs:0,%eax`		
`0x06 addl x@indntpoff,%eax`	`R_386_TLS_IE`	`x`
	Outstanding Relocations	
`GOT[n]`	`R_386_TLS_TPOFF`	`x`

This code sequence results in the same dynamic relocation, but in the instruction it resolves to the absolute address of the GOT slot, not its relative address from the start of GOT.

The GNU variants uses a relocation that computes the negative offset of the variable in the TLS block, rather than the positive offset. This is a significant advantage in that the offset may be embedded directly in a memory address (see below).

Thus to load the contents of `x` (rather than its address) with Sun's model the following code sequence is used.:

Initial Exec Model Code Sequence, IV	**Initial Relocation**	**Symbol**
`0x00 movl x@tpoff(%ebx),%edx`	`R_386_TLS_IE_32`	`x`
`0x06 movl %gs:0,%eax`		
`0x0c subl %edx,%eax`		
`0x0e movl (%eax),%eax`		
	Outstanding Relocations	
`GOT[n]`	`R_386_TLS_TPOFF32`	`x`

This is the same sequence as before with an additional load at the end. In constrast, the GNU sequences don't get longer. The position independent version looks like this:

Initial Exec Model Code Sequence, V	**Initial Relocation**	**Symbol**
`0x00 movl x@gotntpoff(%ebx),%eax`	`R_386_TLS_GOTIE`	`x`
`0x06 movl %gs:(%eax),%eax`		
	Outstanding Relocations	
`GOT[n]`	`R_386_TLS_TPOFF`	`x`

The TLS variant II has the static TLS immediately before the TCB and therefore negative offsets from the memory location pointed to by the `%gs` register directly access it. For the position dependent code the code looks like this:

Initial Exec Model Code Sequence, VI	Initial Relocation	Symbol
`0x00 movl x@indntpoff,%ecx`	`R_386_TLS_IE`	`x`
`0x06 movl %gs:(%ecx),%eax`		
	Outstanding Relocations	
`GOT[n]`	`R_386_TLS_TPOFF`	`x`

In the last sequence, if `%eax` register is used instead of the `%ecx` above, the first instruction may be either 5 or 6 bytes long.

4.3.3 SPARC Initial Exec TLS Model

The SPARC initial exec code sequence given here relies on the GOT pointer in register `%l7` and the thread pointer in register `%g7`. With these registers available the code sequence is simple. We have two different versions, for 32- and 64-bit platforms, since we are loading a GOT entry from memory and this entry differs in size between the 32- and 64-bit machines.

Initial Exec Model Code Sequence, 32-bit	Initial Relocation	Symbol
`0x00 sethi %hi(@tpoff(x)),%o0`	`R_SPARC_TLS_IE_HI22`	`x`
`0x04 or    %o0,%lo(@tpoff(x)),%o0`	`R_SPARC_TLS_IE_LO10`	`x`
`0x08 ld    [%l7+%o0],%o0`	`R_SPARC_TLS_IE_LD`	`x`
`0x0c add   %g7,%o0,%o0`	`R_SPARC_TLS_IE_ADD`	`x`
	Outstanding Relocations, 32-bit	
`GOT[n]`	`R_SPARC_TLS_TPOFF32`	`x`

The code loads the constant offset of the GOT entry in the `%o0` register. The `@tpoff(x)` operator creates the `R_SPARC_TLS_IE_HI22` and `R_SPARC_TLS_IE_LO10` relocations which instruct the linker to allocate the GOT entry and to attach a relocation of type `R_SPARC_TLS_TPOFF32` to it. The `ld` instruction then loads the GOT entry. To allow the linker to recognize the instruction a `R_SPARC_TLS_IE_LD` relocation is added. Finally the `add` instruction computes the address of `x`. The instruction is tagged with a `R_SPARC_TLS_IE_ADD` relocation. Note that the offset generate by the dynamic linker is expected to be negative so that it can be added to the thread pointer.

Initial Exec Model Code Sequence, 64-bit	Initial Relocation	Symbol
`0x00 sethi %hi(@tpoff(x)),%o0`	`R_SPARC_TLS_IE_HI22`	`x`
`0x04 or    %o0,%lo(@tpoff(x)),%o0`	`R_SPARC_TLS_IE_LO10`	`x`

`0x08 ldx   [%l7+%o0],%o0`	R_SPARC_TLS_IE_LDX	x
`0x0c add   %g7,%o0,%o0`	R_SPARC_TLS_IE_ADD	x
	Outstanding Relocations, 64-bit	
`GOT[n]`	R_SPARC_TLS_TPOFF64	x

The 64-bit version is basically identical except that the GOT entry is computed and loaded as a 64-bit value. The relocation used to tag the `ld` instruction also differs accordingly.

4.3.4 SH Initial Exec TLS Model

The initial exec code sequence provides no surprises. It is as simple as one can get it for a RISC machine with the limitations of the small offsets and a not directly usable thread register.

Initial Exec Model Code Sequence	**Initial Relocation Symbol**	
`0x00 mov.l 1f,r0`		
`0x02 stc   gbr,r1`		
`0x04 mov.l @(r0,r12),r0`		
`0x06 bra   2f`		
`0x08  add  r1,r0`		
`     .align 2`		
`1:   .long x@gottpoff`	R_SH_TLS_IE_32	x
`2:   ...`		
	Outstanding Relocations	
`GOT[n]`	R_SH_TLS_TPOFF32	x

The offset of `x` relative to the thread pointer is loaded first. This as usual has to happen indirectly. The word with the label `1:` had the only relocation of the code sequence associate. The linker will fill in the offset of the GOT entry which will contain the offset of the TLS variable in the static TLS block. The GOT entry will be filled by the dynamic linker. The instruction at offset `0x04` loads the value of the GOT entry into register `r0` and then adds the value of the thread register to it. The thread register value is not directly available for an addition so it has to be moved into a regular register first.

For the initial exec code sequence it is once again important that it appears in the output as presented here. The linker has to find the instructions using the relocation generated by `x@gottpoff`.

4.3.5 Alpha Initial Exec TLS Model

The initial exec model requires that the thread pointer be loaded from the PCB into a general purpose register. It is expected that this should be done once at the beginning of the function and the value re-used after that. But for completeness, the PALcall is included in the example sequence.

Initial Exec Model Code Sequence	**Initial Relocation**	**Symbol**
`0x00 call_pal PAL_rduniq`		
`0x04 mov  $0,$tp`		
`     ...`		
`0x10 ldq  $1,x($gp)   !gottprel`	`R_ALPHA_GOTTPREL`	`x`
`0x14 addq $tp,$1,$1`		
	Outstanding Relocations	
`GOT[n]`	`R_ALPHA_TPREL64`	`x`

The `!gottprel` relocation specifier directs the linker to create a GOT entry that contains an associated `R_ALPHA_TPREL64` relocation. This relocation is processed at program startup by the dynamic linker to produce an offset relative to the TCB block for the desired variable. The offset only has to be loaded and added to the value of the thread pointer to obtain the absolute address.

4.3.6 x86-64 Initial Exec TLS Model

The x86-64 initial exec model code uses the `%fs` segment register to locate the TCB. Accessing memory at offset 0 with this segment register enables loading the TCB address.

Initial Exec Model Code Sequence	**Initial Relocation**	**Symbol**
`0x00 movq %fs:0,%rax`		
`0x09 addq x@gottpoff(%rip),%rax`	`R_X86_64_GOTTPOFF`	`x`
	Outstanding Relocations	
`GOT[n]`	`R_X86_64_TPOFF64`	`x`

The assembler generates for the `x@gottpoff(%rip)` expressions a `R_X86_64_GOTTPOFF` relocation for the symbol `x` which requests the linker to generate a GOT entry with a `R_X86_64_TPOFF64` relocation. The offset of the GOT entry relative to the end of the instruction is then used in the instruction. The `R_X86_64_TPOFF64` relocation is processed at program startup time by the dynamic linker by looking up the symbol `x` in the modules loaded at that point. The offset is written in the GOT entry and later loaded by the `addq` instruction.

To load the contents of `x` (rather than its address) an equally long sequence is available:

Initial Exec Model Code Sequence, II	**Initial Relocation**	**Symbol**
`0x00 movq x@gottpoff(%rip),%rax`	`R_X86_64_GOTTPOFF`	`x`
`0x07 movq %fs:(%rax),%rax`		

	Outstanding Relocations	
`GOT[n]`	`R_X86_64_TPOFF64`	`x`

4.3.7 s390 Initial Exec TLS Model

The code for the initial exec model is small and fast. The code has to get the offset relative to the thread pointer from the GOT and add it to the thread pointer. There are three different variants. The position independent variant with a small GOT (`-fpic`) is:

Initial Exec Model Code Sequence	**Initial Relocation**	**Symbol**
`    ear %r7,%a0`		
`    l   %r9,x@gotntpoff(%r12)`	`R_390_TLS_GOTIE12`	`x`
`    la  %r10,0(%r9,%r7) # %r10 = &x`		
	Outstanding Relocations	
`GOT[n]`	`R_390_TLS_TPOFF32`	`x`

The `R_390_TLS_GOTIE12` relocation created for the expression `x@gotntpoff` causes the linker to generate a GOT entry with a `R_390_TLS_TPOFF` relocation. `x@gotntpoff` is replaced by the linker with the 12 bit offset from the start of the GOT to the generated GOT entry. The `R_390_TLS_TPOFF` relocation is processed at program startup time by the dynamic linker.

The position independent variant with a large GOT (`-fPIC`) is:

Initial Exec Model Code Sequence	**Initial Relocation**	**Symbol**
`    ear %r7,%a0`		
`    l   %r8,.L1-.L0(%r13)`		
`    l   %r9,0(%r8,%r12)`	`R_390_TLS_LOAD`	`x`
`    la  %r10,0(%r9,%r7) # %r10 = &x`		
`    ...`		
`.L0: # literal pool, address in %r13`		
`.L1: .long x@gotntpoff`	`R_390_TLS_GOTIE32`	`x`
	Outstanding Relocations	
`GOT[n]`	`R_390_TLS_TPOFF32`	`x`

The `R_390_TLS_GOTIE32` relocation does the same as `R_390_TLS_GOTIE12`, the difference is that the linker replaces the `x@gotntpoff` expression with a 32 bit GOT offset instead of 12 bit.

The variant without GOT pointer is:

Initial Exec Model Code Sequence	Initial Relocation	Symbol
`      ear %r7,%a0`		
`      l   %r8,.L1-.L0(%r13)`		
`      l   %r9,0(%r8)`	`R_390_TLS_LOAD`	`x`
`      la  %r10,0(%r9,%r7) # %r10 = &x`		
`      ...`		
`.L0: # literal pool, address in %r13`		
`.L1: .long x@indntpoff`	`R_390_TLS_IE32`	`x`
	Outstanding Relocations	
`GOT[n]`	`R_390_TLS_TPOFF32`	`x`

The `R_390_TLS_IE32` relocation instructs the linker to create the same GOT entry as for `R_390_TLS_GOTIE`{`12,32`} but the linker replaces the `x@indntpoff` expression with the absolute address of the created GOT entry. This makes the variant without GOT pointer inadequate for position independent code.

4.3.8 s390x Initial Exec TLS Model

The initial exec model for s390x works like the initial exec model for s390. The position independent variant with a small GOT (`-fpic`) is:

Initial Exec Model Code Sequence	Initial Relocation	Symbol
`      ear  %r7,%a0`		
`      sllg %r7,%r7,32`		
`      ear  %r7,%a1`		
`      lg   %r9,x@gotntpoff(%r12)`	`R_390_TLS_GOTIE12`	`x`
`      la   %r10,0(%r9,%r7) # %r10 = &x`		
	Outstanding Relocations	
`GOT[n]`	`R_390_TLS_TPOFF32`	`x`

The position independent variant with a large GOT (`-fPIC`) is:

Initial Exec Model Code Sequence	Initial Relocation	Symbol

`ear  %r7,%a0`		
`sllg %r7,%r7,32`		
`ear  %r7,%a1`		
`lg   %r8,.L1-.L0(%r13)`		
`lg   %r9,0(%r8,%r12)`	`R_390_TLS_LOAD`	`x`
`la   %r10,0(%r9,%r7) # %r10 = &x`		
`...`		
`.L0: # literal pool, address in %r13`		
`.L1: .quad x@gotntpoff`	`R_390_TLS_GOTIE64`	`x`
	Outstanding Relocations	
`GOT[n]`	`R_390_TLS_TPOFF64`	`x`

The linker will replace `x@gotntpoff` for `R_390_TLS_GOTIE64` with a 64 bit GOT offset. The variant without GOT pointer is:

Initial Exec Model Code Sequence	**Initial Relocation**	**Symbol**
`ear  %r7,%a0`		
`sllg %r7,%r7,32`		
`ear  %r7,%a1`		
`larl %r8,x@indntpoff`	`R_390_TLS_IEENT`	`x`
`lg   %r9,0(%r8)`	`R_390_TLS_LOAD`	`x`
`la   %r10,0(%r9,%r7) # %r10 = &x`		
	Outstanding Relocations	
`GOT[n]`	`R_390_TLS_TPOFF64`	`x`

The `R_390_TLS_IEENT` relocations causes `x@indntpoff` to be replaced with the relative offset from the `larl` instruction to the GOT entry. Because the instruction is pc relative the variant without GOT pointer can be used in position independent code as well.

4.4 Local Exec TLS Model

Optimizations for the local dynamic model, similar to those the local dynamic model adds to the generic dynamic model, lead to the local exec model. Its use is even more restricted than that of the local dynamic model. It can only be used for code in the executable itself and to access variables in the executable itself.

Restricting the use to the executable means that just as for the local exec model that the TLS block can be addressed relative to the thread pointer. Restricting the variables to only those defined in the executable means that always the first TLS block, the one for the executable, is used and therefore the size of all the other TLS blocks is irrelevant for the address computation. It also means that the linker knows when creating the final executable what the offset from the TCB is. The formula for the actual offset depends on the architecture but it consists of a sum or difference of the thread pointer, the offset of the first TLS block $tlsoffset_1$ and the offset of the variable in this TLS block $offset_x$. The result is known at link-time and is made available in the code as an immediate value.

The code in the architecture descriptions in the next sections implements something along the line of the following where the code must be in the executable itself:

```
static __thread int x;

&x;
```

4.4.1 IA-64 Local Exec TLS Model

The code sequence for this model is very simple. If the thread register value is maintained appropriately in a register suitable for the `add` instruction the code sequence consists of only one instructions for every new variable.

Local Exec Model Code Sequence	Initial Relocation	Symbol
`0x00 ld8  r2=tp`		
`     ;;`		
`0x10 addl loc0=@tprel(x),r2`	`R_IA_64_TPREL22`	`x`
	Outstanding Relocations	

Beside preparing the `add` instruction by moving the thread pointer value in the `r2` register all the code does is adding the constant offset to the thread pointer (the `add` instruction cannot directly use the `tp` register). The `R_IA_64_TPREL22` relocation names the variable and the linker is performing determining $tlsoffset_1 + offset_x$. I.e., beside the offset of the variable in the TLS block only the alignment of the TLS block has an influence on the result.

As with the initial exec model the code sequence given here is one of three possible one. It allows handling of thread-local data up to 2^{21} bytes (2 MiBi). Optimization are possible for dealing with less than 2^{13} bytes (8 KiBi) or more then 2^{21} bytes in which case the relocations used are `R_IA_64_TPREL14` and `R_IA_64_TPREL64I` respectively and the instruction is either a short add or a long move.

4.4.2 IA-32 Local Exec TLS Model

The IA-32 code sequence basically is only an addition of the offset which is available as an immediate value to the thread pointer. The way the thread pointer is determined might vary; in Sun's model it can be determined by loading at offset 0 from the `%gs` segment.

Local Exec Model Code Sequence	Initial Relocation	Symbol
`0x00 movl $x@tpoff,%edx`	R_386_TLS_LE_32	x1
`0x05 movl %gs:0,%eax`		
`0x0b subl %edx,%eax`		
	Outstanding Relocations	

The x@tpoff expression is used here not as an offset relative to the GOT but instead as an immediate value. For this the linker generates a R_386_TLS_LE_32 relocation which can be resolved by the linker. The value so determined is the positive offset of the variable in the TLS block. It is subtracted from the thread pointer value to lead to the final address of x in the %eax register. The GNU variant has again the advantage of being shorter.

Local Exec Model Code Sequence, II	Initial Relocation	Symbol
`0x00 movl %gs:0,%eax`		
`0x06 leal x@ntpoff(%eax),%eax`	R_386_TLS_LE	x
	Outstanding Relocations	

Here the GNU variant uses a relocation that computes the negative offset of the variable in the TLS block, rather than the positive offset. This is a significant advantage in that the offset may be embedded directly in a memory address (see below).

Thus to load the contents of x (rather than its address) with Sun's model the following code sequence is used:

Local Exec Model Code Sequence III	Initial Relocation	Symbol
`0x00 movl $x@tpoff,%edx`	R_386_TLS_LE_32	x1
`0x05 movl %gs:0,%eax`		
`0x0b subl %edx,%eax`		
`0x0d movl (%eax),%eax`		
	Outstanding Relocations	

This is the same sequence as before with an additional load at the end. In contrast, the GNU sequence does not get longer:

Local Exec Model Code Sequence, IV	Initial Relocation	Symbol

`0x00 movl %gs:0,%eax`
`0x06 movl x@ntpoff(%eax),%eax`
	R_386_TLS_LE	x

	Outstanding Relocations	

If instead of computing the address of the variable we want to load from it or store in it the following "sequence" can be used. Note that in this case we use the `x@ntpoff` expression not as an immediate value but instead as an absolute address.

Local Exec Model Code Sequence, V	**Initial Relocation**	**Symbol**
`0x00 movl %gs:x@ntpoff,%eax`	R_386_TLS_LE	x
	Outstanding Relocations	

The fact that the load and store operation is even simpler than the computation of the address is certainly astonishing at first. But the segment register handling is weird. One can think of the segment register `%gs` as a mean to move the zero address of the virtual address space to a different location. The new location once computed is directly accessible only to the CPU internals. This is why computing its address at user-level requires the additional requirement that the first word of the shifted address space contain the shift value or address.

4.4.3 SPARC Local Exec TLS Model

The SPARC local exec model code sequence is as easy as can get. It is just a matter of adding the offset, which is available as an immediate value, to the thread register value.

Local Exec Model Code Sequence	**Initial Relocation**	**Symbol**
`0x00 sethi %hix(@tpoff(x)),%o0`	R_SPARC_TLS_LE_HIX22	x
`0x04 xor   %o0,%lox(@tpoff(x)),%o0`	R_SPARC_TLS_LE_LOX10	x
`0x08 add   %g7,%o0,%o0`		
	Outstanding Relocations	

The `%hix(tpoff(x))` and `%lox(tpoff(x))` expressions cause the assembler to emit the `R_SPARC_TLS_LE_HIX22` and `R_SPARC_TLS_LE_LOX10` relocations which request the linker to fill the offset value in the instructions as immediate values. This loads the offset into the `%o0` register. The following `add` instruction requires that the offset here is negative. To compute the final address the offset is added to the value of the thread register `%g7`. The `add` instruction is **not** tagged with a relocation. The reason is that the linker will never have to recognize this instruction for relaxation since it does not get any simpler.

4.4.4 SH Local Exec TLS Model

As for the other architectures the local exec model code sequence is really simple. The main difference is that as for all SH code a data relocation is needed.

Local Exec Model Code Sequence	Initial Relocation	Symbol
`0x00 mov.l .Ln,r0`		
`0x02 stc   gbr,r1`		
`0x04  add  r1,r0`		
`     ...`		
`.Ln: .long x@tpoff`	`R_SH_TLS_LE_32`	`x`
	Outstanding Relocations	

This code loads the two components of the address, the thread-pointer relative offset (known at linktime) and the thread pointer, in the registers `r0` and `r1` respectively and adds them. Since no more optimzation is possible from this code sequence the exact location of the word with the label `.Ln` is unimportant.

4.4.5 Alpha Local Exec TLS Model

The Alpha local exec model sequences are nice and tidy. There are three sequences to choose from, depending on the size of the TLS that the application expects. In the sequences below, it is expected that `PAL_rduniq` has been invoked, and the thread pointer copied to `$tp`.

Local Exec Model Code Sequence	Initial Relocation	Symbol
`0x00 lda  $1,x1($tp)   !tprel`	`R_ALPHA_TPREL16`	`x1`
`     ...`		
`0x10 ldah $1,x2($tp)   !tprelhi`	`R_ALPHA_TPRELHI`	`x2`
`0x14 lda  $1,x2($1)    !tprello`	`R_ALPHA_TPRELLO`	`x2`
`     ...`		
`0x20 ldq  $1,x3($gp)   !gottprel`	`R_ALPHA_GOTTPREL`	`x3`
`0x24 add1 $1,$tp,$1`		
	Outstanding Relocations	

The first sequence is good for 32 KiB, the second sequence for 2 GiB, and the third for a full 64 bit displacement.

4.4.6 x86-64 Local Exec TLS Model

The x86-64 code sequence is similar to IA-32 GNU variant. It is only an addition of the offset which is available as an immediate value to the thread pointer. The thread pointer is loaded from offset 0 of the `%fs` segment.

Local Exec Model Code Sequence	Initial Relocation	Symbol
`0x00 movq %fs:0,%rax`		
`0x09 leaq x@tpoff(%rax),%rax`	R_X86_64_TPOFF32	x
	Outstanding Relocations	

To load a TLS variable instead of computing its address, the following sequence can be used:

Local Exec Model Code Sequence, II	Initial Relocation	Symbol
`0x00 movq %fs:0,%rax`		
`0x09 movq x@tpoff(%rax),%rax`	R_X86_64_TPOFF32	x
	Outstanding Relocations	

or shorter:

Local Exec Model Code Sequence, III	Initial Relocation	Symbol
`0x00 movq %fs:x@tpoff,%rax`	R_X86_64_TPOFF32	x
	Outstanding Relocations	

4.4.7 s390 Local Exec TLS Model

The local exec model for s390 is only an addition of the offset which is available as an immediate value to the thread pointer. In general the offset can have 32 bit which requires a literal pool entry.

Local Exec Model Code Sequence	Initial Relocation	Symbol
`ear %r7,%a0`		
`l   %r8,.L1-.L0(%r13)`		

`        la  %r9,0(%r8,%r7) # %r9 = &x`		
`        ...`		
`.L0: # literal pool, address in %r13`		
`.L1: .long x@ntpoff`	`R_390_TLS_LE32`	`x`
	Outstanding Relocations	

The linker resolves the `R_390_TLS_LE32` relocation to a negative offset to the thread pointer.

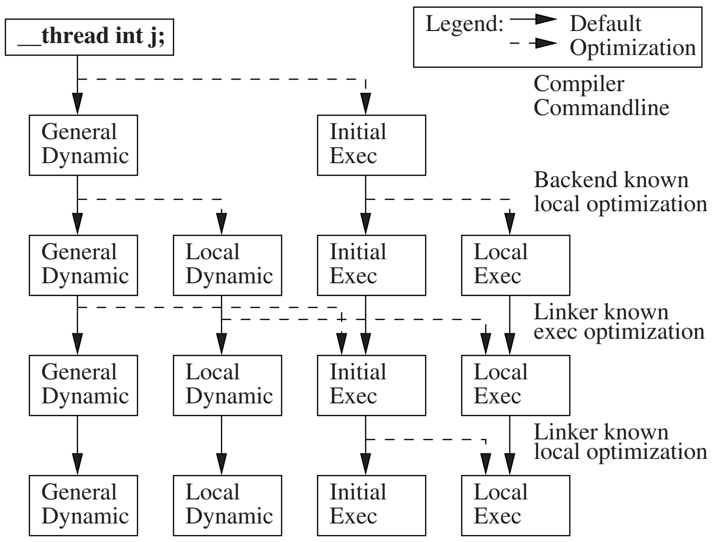
4.4.8 s390x Local Exec TLS Model

The local exec model for s390x differs to the s390 model only in the thread pointer extraction and the size of the offset.

Local Exec Model Code Sequence	**Initial Relocation**	**Symbol**
`        ear  %r7,%a0`		
`        sllg %r7,%r7,32`		
`        ear  %r7,%a1`		
`        lg   %r8,.L1-.L0(%r13)`		
`        la   %r9,0(%r8,%r7) # %r9 = &x`		
`        ...`		
`.L0: # literal pool, address in %r13`		
`.L1: .quad x@ntpoff`	`R_390_TLS_LE64`	`x`
	Outstanding Relocations	

5 Linker Optimizations

The thread-local storage access model are hierarchical in the way they can be used. The most generic model is the general dynamic model which can be used everywhere. The initial exec model can be used unconditionally when generating the executable itself. It can also be used if a shared object is not meant to be dynamically loaded. These two models already define a hierarchy. The other two models are special optimizations for either one of the more generic models if the definition is in the same module as the reference. Graphically the hierarchy and transitions between the access models can be represented like this:[4]



The diagram shows how a code sequence to access a thread-local variable can be optimized (or not) by compiler and linker. The solid lines indicate the default path taken from any position. The default is to always leave the code as it is. Optimization are indicated by the dashed lines.

Optimizations can have five different reasons:

- The programmer tells the compiler that the generated code is for an executable and not used in a shared object.
- The programmer tells the compiler that the generated code does not have to access variables in dynamically loaded code directly (using `dlsym` is OK).
- The compiler realizes that a thread-local variable is protected. I.e., the reference is in the same module as the definition.
- The linker knows whether an executable (type `ET_EXEC`) is created or an shared object (type `ET_DYN`).

[4]This nice illustration was originally developed by Mike Walker.

- The linker knows whether a reference to thread-local variable from code in the executable is unconditionally satisfied by a definition in the executable itself. The definition need not be protected since the executable is always the first object in the symbol lookup path.

In the description of the access models for the architectures we already explained the prerequisites for the use of the model. In the following section we explain in detail how the code relaxations have to happen. This is not exactly trivial since there is not a 1:1 relationship between the instructions used in the code sequences and we have to handle differences in the sizes.

Of the architectures defined in this document so far only IA-32, SPARC, x86-64, Alpha, and SH have defined linker optimizations. Doing this for IA-64 would be very difficult to say the least. Code generation for IA-64 ideally has to move the bundles given in the code sequences as far away from each other as possible to increase parallelism. But this means that locating the instructions which belong together is everything but trivial. Not even tagging the instructions with relocations would work since multiple code sequences could be merged together to load from or store in multiple thread-local variables at once. Only very complicated flow analysis could reveal the individual code sequences and nothing like this is current planned.

The architectures which define optimizations require that the compiler emits code sequences as described. This, together with the relocations tagging the instructions, will allow the linker to recognize the code sequences. Minor variations like using different registers can easily be masked out. The details of how the code sequences are recognized will not be discussed here. We assume that the linker has the capabilities and concentrate on the actual work which has to be done now.

5.1 IA-32 Linker Optimizations

The linker is able to perform four different optimizations which save execution time by reducing run-time relocations and loads from memory. The diagram only shows three transitions but the initial exec to local exec transformation can be performed in addition to others. Since the code sequences for the Sun and GNU variants are different we need to discuss them here separately as well.

One word on the side effect of some of the optimizations. If the original code uses the general dynamic or local dynamic access model the `__tls_get_addr` function is used to access the variables. If none but these two models is used this means that the allocation of the TLS blocks can be deferred as explained in the previous sections. If the linker performs its optimizations access to the TLS block happens without `__tls_get_addr` getting the chance to eventually allocate the memory the static model is automatically enabled and the `DF_STATIC_TLS` flag must be set. This is normally not a deterrent since the access to the static TLS block is frequent and deferred allocation is really most useful for dynamically loaded code.

General Dynamic To Initial Exec

Probably the most important of the relaxations the linker can perform is the change from the general dynamic to the initial exec model. The general dynamic model is the most expensive at run-time and therefore should be avoided if possible. First we handle the Sun variant.

GD ⟶ IE Code Transition	Initial Relocation	Symbol
`0x00 leal  x@dtlndx(%ebx),%edx`	R_386_TLS_GD_32	x
`0x06 pushl %edx`	R_386_TLS_GD_PUSH	x
`0x07 call  x@TLSPLT`	R_386_TLS_GD_CALL	x
`0x0c popl  %edx`	R_386_TLS_GD_POP	x
`0x0d nop`		
⇓	⇓	⇓
`0x00 movl  x@tpoff(%ebx),%edx`	R_386_TLS_IE_32	x
`0x06 movl  %gs:0,%eax`		
`0x0c subl  %edx,%eax`		
	Outstanding Relocations	
`GOT[n]`	R_386_TLS_TPOFF32	x

This optimization can be performed whenever an executable is created. The optimization for the GNU variant is similar:

GD ⟶ IE Code Transition	Initial Relocation	Symbol
`0x00 leal x@tlsgd(,%ebx,1),%eax`	R_386_TLS_GD	x
`0x07 call ___tls_get_addr@plt`	R_386_PLT32	___tls_get_addr
⇓	⇓	⇓
`0x00 movl %gs:0,%eax`		
`0x06 addl x@gotntpoff(%ebx),%eax`	R_386_TLS_GOTIE	x
	Outstanding Relocations	
`GOT[n]`	R_386_TLS_TPOFF	x

It should now be clear why the general dynamic model code sequences for both variants are longer than necessary. The `nop` in Sun's case and the use of the SIB-form in the GNU variant are needed to have room for the IE code sequence.

General Dynamic To Local Exec

The symbol lookup rules for ELF define that if a symbol needed in the executable is defined in the executable it is always picked. The reason is that the executable is always at the head of the search scope list. Therefore the general dynamic to local exec is quite frequent as well and can save even more than the transition to the initial exec model.

GD → LE Code Transition	Initial Relocation	Symbol
`0x00 leal  x@dtlndx(%ebx),%edx`	R_386_TLS_GD_32	x
`0x06 pushl %edx`	R_386_TLS_GD_PUSH	x
`0x07 call  x@TLSPLT`	R_386_TLS_GD_CALL	x
`0x0c popl  %edx`	R_386_TLS_GD_POP	x
`0x0d nop`		
⇓	⇓	⇓
`0x00 movl  $x@tpoff,%edx`	R_386_TLS_LE_32	x
`0x05 nop`		
`0x06 movl  %gs:0,%eax`		
`0x0c subl  %edx,%eax`		
	Outstanding Relocations	

This optimization for the Sun variant reduces the number of instructions by one and replaces the function call with a memory load and an arithmetic operation. The GNU variant is equally effective:

GD → LE Code Transition	Initial Relocation	Symbol
`0x00 leal x@tlsgd(,%ebx,1),%eax`	R_386_TLS_GD	x
`0x07 call ___tls_get_addr@plt`	R_386_PLT32	___tls_get_addr
⇓	⇓	⇓
`0x00 movl %gs:0,%eax`		
`0x06 addl $x@ntpoff,%eax`	R_386_TLS_LE	x
	Outstanding Relocations	

Please note the length of the `movl` instruction in the replacement code. It assumes that a modR/M byte is used.

Local Dynamic to Local Exec

If the user did not tell the compiler that the code is intended for an executable it is still possible for the linker to optimize the code but as can be seen below, the result is not optimal.

LD → LE Code Transition	Initial Relocation	Symbol
`0x00 leal  x1@tmdnx(%ebx),%edx`	R_386_TLS_LDM_32	x1
`0x06 pushl %edx`	R_386_TLS_LDM_PUSH	x1
`0x07 call  x1@TLSPLT`	R_386_TLS_LDM_CALL	x1
`0x0c popl  %edx`	R_386_TLS_LDM_POP	x1
`     ...`		
`0x10 movl  $x1@dtpoff,%edx`	R_386_TLS_LDO_32	x1

0x15 addl %eax,%edx		
⇓	⇓	⇓
0x00 movl %gs:0,%eax		
0x06 nop		
0x07 nop		
0x08 nop		
0x09 nop		
0x0a nop		
0x0b nop		
0x0c nop		
...		
0x10 movl $x1@tpoff,%eax	R_386_TLS_LE_32	x1
0x15 addl %eax,%edx		
	Outstanding Relocations	

The long sequence of nops is the result of the large code size for the code sequence generated for the local dynamic model. It is unavoidable at this point. Only the programmer telling the compiler that the code is for an executable could have avoided it. What is described here is what Sun documents. The GNU variant has to same problem but solves it with a bit less negative impact on run-time performance.

LD ⟶ LE Code Transition	**Initial Relocation**	**Symbol**
0x00 leal x1@tlsldm(%ebx),%eax	R_386_TLS_LDM	x1
0x06 call ___tls_get_addr@plt	R_386_PLT32	___tls_get_addr
...		
0x10 leal x1@dtpoff(%eax),%edx	R_386_TLS_LDO_32	x1
⇓	⇓	⇓
0x00 movl %gs:0,%eax		
0x06 nop		
0x07 leal 0x0(%esi,1),%esi		
...		
0x10 leal x1@ntpoff(%eax),%edx	R_386_TLS_LE	x1
	Outstanding Relocations	

The instruction at address `0x07` requires some explanation. It might look some pretty expensive instruction which does a lot but in fact it is a no-op. The value of the `%esi` register is stored in the same register after multiplying it with one and adding zero. The reason this instruction is chosen is that it is long, 4 bytes to be exact. This means to fill the 5 byte hole we only need one extra `nop` instruction. This is much cheaper than using seven `nop` instructions (similar to what Sun does).

In case the local dynamic model code is not computing the address and instead loads from or stores in the variable directly the transformed code is also simply loading or storing. The transformation is simple and just as documented in the example code above: replace the `x1@dtpoff(%eax)` expression with `-x1@tpoff(%eax)` which is accomplished by changing the `R_386_TLS_LDO_32` relocation into a `R_386_TLS_LE_32` relocation.

Initial Exec To Local Exec

The last optimization helps to squeeze out the last bit of performance if the code was already compiled for exclusive use in an executable and a variable was found to be available in the executable itself. This transition is much less wasteful than the local dynamic to local exec transition.

IE → LE Code Transition	**Initial Relocation**	**Symbol**
`0x00 movl x@tpoff(%ebx),%edx`	`R_386_TLS_IE_32`	`x`
`0x06 movl %gs:0,%eax`		
`0x0c subl %edx,%eax`		
⇓	⇓	⇓
`0x00 movl $x@tpoff,%edx`	`R_386_TLS_LE_32`	`x`
`0x05 nop`		
`0x06 movl %gs:0,%eax`		
`0x0c subl %edx,%eax`		
	Outstanding Relocations	

This optimization saves one run-time relocation, transforms one memory load into a load of an immediate value but also adds a new instruction. This instruction is a `nop` and which does not disrupt the execution much. The GNU variant does not need such ugliness:

IE → LE Code Transition	**Initial Relocation**	**Symbol**
`0x00 movl %gs:0,%eax`		
`0x06 addl x@gotntpoff(%ebx),%eax`	`R_386_TLS_GOTIE`	`x`
⇓	⇓	⇓
`0x00 movl %gs:0,%eax`		
`0x06 leal x@ntpoff(%eax),%eax`	`R_386_TLS_LE`	`x`
	Outstanding Relocations	

5.2 SPARC Linker Optimizations

Since the model used for SPARC is mostly identical to that of IA-32 it is not surprising that the same four optimizations are available here as well. In general the optimization are a bit cleaner due to the RISC instruction set of the SPARC processor which unlike the CISC of IA-32 has a uniform length for the instructions.

General Dynamic To Initial Exec

This optimization manages to get rid of one run-time relocation and the call to the `__tls_get_addr` function. But the memory allocation for the static TLS block cannot be deferred anymore and the `DF_STATIC_TLS` flag must be set.

GD → IE Code Transition, 32-bit	**Initial Relocation**	**Symbol**
`0x00 sethi %hi(@dtlndx(x)),%o0`	R_SPARC_TLS_GD_HI22	x
`0x04 add   %o0,%lo(@dtlndx(x)),%o0`	R_SPARC_TLS_GD_LO10	x
`0x08 add   %l7,%o0,%o0`	R_SPARC_TLS_GD_ADD	x
`0x0c call  __tls_get_addr`	R_SPARC_TLS_GD_CALL	x
⇓	⇓	⇓
`0x00 sethi %hi(@tpoff(x)),%o0`	R_SPARC_TLS_IE_HI22	x
`0x04 or    %o0,%lo(@tpoff(x)),%o0`	R_SPARC_TLS_IE_LO10	x
`0x08 ld    [%l7+%o0],%o0`		
`0x0c add   %g7,%o0,%o0`		
	Outstanding Relocations	
`GOT[n]`	R_SPARC_TLS_TPOFF32	x

We do not list the 64-bit version here as well. The differences are the same as described in section 4.3.3. The actual register used for the GOT pointer (`%l7` in the code above) can vary. The linker will figure the actual register used out from the instruction tagged with `R_SPARC_TLS_GD_ADD`.

General Dynamic To Local Exec

This optimization is also straight-forward, the instructions of the general dynamic model are simply replaced by those of the local exec model. The only thing to keep in mind is filling the short local exec code sequence with a `nop`.

GD → LE Code Transition	**Initial Relocation**	**Symbol**
`0x00 sethi %hi(@dtlndx(x)),%o0`	R_SPARC_TLS_GD_HI22	x
`0x04 add   %o0,%lo(@dtlndx(x)),%o0`	R_SPARC_TLS_GD_LO10	x
`0x08 add   %l7,%o0,%o0`	R_SPARC_TLS_GD_ADD	x
`0x0c call  __tls_get_addr`	R_SPARC_TLS_GD_CALL	x
⇓	⇓	⇓
`0x00 sethi %hix(@tpoff(x)),%o0`	R_SPARC_TLS_LE_HIX22	x
`0x04 xor   %o0,%lox(@tpoff(x)),%o0`	R_SPARC_TLS_LE_LOX10	x
`0x08 add   %g7,%o0,%o0`		
`0x0c nop`		
	Outstanding Relocations	

This optimization removes both run-time relocations and the call to the `__tls_get_addr` function. The executable must be marked with the `DF_STATIC_TLS` flag, though.

Local Dynamic To Local Exec

The transition from the local dynamic to the local exec model is also on SPARC the least optimal. It is best to enable the compiler to generate the optimal code right away. But the optimization is nevertheless effective since it eliminates one run-time relocation and the call to the `__tls_get_addr` function.

LD ⟶ LE Code Transition	**Initial Relocation**	**Symbol**
`0x00 sethi %hi(@tmdnx(x1)),%o0`	`R_SPARC_TLS_LDM_HI22`	`x1`
`0x04 add   %o0,%lo(@tmndx(x1)),%o0`	`R_SPARC_TLS_LDM_LO10`	`x1`
`0x08 add   %l7,%o0,%o0`	`R_SPARC_TLS_LDM_ADD`	`x1`
`0x0c call  __tls_get_addr`	`R_SPARC_TLS_LDM_CALL`	`x1`
`     ...`		
`0x10 sethi %hix(@dtpoff(x1)),%l1`	`R_SPARC_TLS_LDO_HIX22`	`x1`
`0x14 xor   %l1,%lox(@dtpoff(x1)),%l1`	`R_SPARC_TLS_LDO_LOX22`	`x1`
`0x18 add   %o0,%l1,%l1`	`R_SPARC_TLS_LDO_ADD`	`x1`
⇓	⇓	⇓
`0x00 nop`		
`0x04 nop`		
`0x08 nop`		
`0x0c mov   %g0, %o0`		
`     ...`		
`0x10 sethi %hix(@tpoff(x1)),%o0`	`R_SPARC_TLS_LE_HIX22`	`x1`
`0x14 xor   %o0,%lox(@tpoff(x1)),%o0`	`R_SPARC_TLS_LE_LOX10`	`x1`
`0x18 add   %g7,%o0,%o0`		
	Outstanding Relocations	

This optimization also requires that the executable is marked with the `DF_STATIC_TLS` flag.

Initial Exec To Local Exec

If the programmer told the compiler the code is meant for the executable but only the linker knows that a variable is defined in the executable itself the following optimization helps to eliminate the remaining run-time relocation.

IE ⟶ LE Code Transition	**Initial Relocation**	**Symbol**
`0x00 sethi %hi(@tpoff(x)),%o0`	`R_SPARC_TLS_IE_HI22`	`x`
`0x04 or    %o0,%lo(@tpoff(x)),%o0`	`R_SPARC_TLS_IE_LO10`	`x`
`0x08 ld    [%l7+%o0],%o0`	`R_SPARC_TLS_IE_LD`	`x`
`0x0c add   %g7,%o0,%o0`	`R_SPARC_TLS_IE_ADD`	`x`
⇓	⇓	⇓
`0x00 sethi %hix(@tpoff(x)),%o0`	`R_SPARC_TLS_LE_HIX22`	`x`
`0x04 xor   %o0,%lox(@tpoff(x)),%o0`	`R_SPARC_TLS_LE_LOX10`	`x`
`0x08 mov   %o0,%o0`		

`0x0c add   %g7,%o0,%o0`	
	Outstanding Relocations

Since the local exec model code sequence has only three instructions the instruction at address `0x08` is added as a no-op.

5.3 SH Linker Optimizations

As for IA-32 and SPARC the linker can perform a number of optimizations. But the repertoire is limited due to the structure of the SH code and the code sequences used.

General Dynamic to Initial Exec

If the initial exec model can be used code compiled using the general dynamic model can save two instructions and potentially one GOT entry by performing the following transformation:

GD $\longrightarrow$ IE Code Transition	**Initial Relocation**	**Symbol**
`0x00 mov.l 1f,r4`		
`0x02 mova  2f,r0`		
`0x04 mov.l 2f,r1`		
`0x06 add   r0,r1`		
`0x08 jsr   @r1`		
`0x0a  add  r12,r4`		
`0x0c bra   3f`		
`0x0e  nop`		
`     .align 2`		
`1:   .long x@tlsgd`	R_SH_TLS_GD_32	x
`2:   .long __tls_get_addr@plt`		
`3:`		
⇓	⇓	⇓
`0x00 mov.l 1f,r0`		
`0x02 stc   gbr,r4`		
`0x04 mov.l @(r0,r12),r0`		
`0x06 bra   3f`		
`0x08  add  r4,r0`		
`0x0a nop`		
`0x0c nop`		
`0x0e nop`		
`     .align 2`		
`1:   .long x@gottpoff`	R_SH_TLS_IE_32	x
`2:   .long 0`		
`3:`		
	Outstanding Relocations	
`GOT[n]`	R_SH_TLS_TPOFF32	x

The call to `__tls_get_addr` has been optimized out and the instructions and the data definition associated with the jump are complete gone. Note that we can move the `bra` instruction for so that the now unnecessary memory locations filled with `nop` are never executed.

General Dynamic to Local Exec

The transformation from general dynamic to the local exec model is almost identical to the last transformation. Only we save one more instruction and there is no dynamic relocation left.

GD ⟶ IE Code Transition	**Initial Relocation**	**Symbol**
`0x00 mov.l 1f,r4`		
`0x02 mova  2f,r0`		
`0x04 mov.l 2f,r1`		
`0x06 add   r0,r1`		
`0x08 jsr   @r1`		
`0x0a  add  r12,r4`		
`0x0c bra   3f`		
`0x0e  nop`		
`     .align 2`		
`1:   .long x@tlsgd`	R_SH_TLS_GD_32	x
`2:   .long __tls_get_addr@plt`		
`3:`		
⇓	⇓	⇓
`0x00 mov.l 1f,r0`		
`0x02 stc   gbr,r4`		
`0x04 bra   3f`		
`0x06  add  r4,r0`		
`0x08 nop`		
`0x0a nop`		
`0x0c nop`		
`0x0e nop`		
`     .align 2`		
`1:   .long x@tpoff`	R_SH_TLS_LE_32	x
`2:   .long 0`		
`3:`		
	Outstanding Relocations	

Again it is possible to place the `bra` instruction tactically well to avoid having the execute all the `nop` instructions which have to be filled in.

Local Dynamic to Local Exec

The final optimization allows converting local dynamic code sequences to locale exec code sequences. The code generation this way is potentially even more efficient than the local exec code sequence described above since the thread register is read only once.

Local Dynamic Model Code Sequence	**Initial Relocation**	**Symbol**

```
0x00 mov.l 1f,r4                   |
0x02 mova  2f,r0                   |
0x04 mov.l 2f,r1                   |
0x06 add   r0,r1                   |
0x08 jsr   @r1                     |
0x0a  add  r12,r4                  |
0x0c bra   3f                      |
0x0e  nop                          |
     .align 2                      |
1:   .long x1@tlsgd                | R_SH_TLS_LD_32     x1
2:   .long __tls_get_addr@plt      |
3:   ...                           |
     mov.l .Lp,r1                  |
     mov.l r0,r1                   |
     ...                           |
     mov.l .Lq,r1                  |
     mov.l r0,r1                   |
     ...                           |
.Lp: .long x1@dtpoff               | R_SH_TLS_LDO_32    x1
.Lp: .long x2@dtpoff               | R_SH_TLS_LDO_32    x2
                ⇓                          ⇓           ⇓
0x00 bra   3f                      |
0x02  stc  gbr,r0                  |
0x04 nop                           |
0x06 nop                           |
0x08 nop                           |
0x0a nop                           |
0x0c nop                           |
0x0e nop                           |
     .align 2                      |
1:   .long 0                       |
2:   .long 0                       |
3:   ...                           |
     mov.l .Lp,r1                  |
     mov.l r0,r1                   |
     ...                           |
     mov.l .Lq,r1                  |
     mov.l r0,r1                   |
     ...                           |
.Lp: .long x1@tpoff                | R_SH_TLS_LE_32     x1
.Lp: .long x2@tpoff                | R_SH_TLS_LE_32     x2
                                   | Outstanding Relocations
```

Since computing the base address for the relocation used is now very simple (just loading the thead register) the prologue is almost empty. The one instruction can be executed in the branch delay slot of the jump over the first data.

5.4 Alpha Linker Optimizations

The Alpha linker optimizations are cleaner than either the IA-32 or SPARC, because there are no restrictions on the ordering of instructions.

The `TLSGD`/`TLSLDM`, `LITERAL`, and `LITUSE` relocations are related by sequence number in the assembly file. This causes them to be emitted adjacent into the object file.

Relaxation of the `__tls_get_addr` patterns cannot occur unless relocations `TLSGD`, `LITERAL`, and `LITUSE_TLSGD` appear in that exact sequence (and similar for `TLSLDM`). This is to distinguish the case where the `TLSGD` relocation is not associated with any one call sequence. The assembler will enforce the constraint that if `LITUSE_TLSGD` exists, the `TLSGD` and `LITERAL` relocations will be present, and no other `LITUSE` relocations will be associated with the `LITERAL`.

Relaxation of the `__tls_get_addr` patterns require that there be a `GPDISP` relocation at the offset immediately following the `jsr`.

General Dynamic To Initial Exec

GD ⟶ IE Code Transition	Initial Relocation	Symbol
`0x00 lda  $16,x($gp)`	`R_ALPHA_TLSGD`	`x`
`0x04 ldq  $27,__tls_get_addr($gp)`	`R_ALPHA_LITERAL`	`__tls_get_addr`
`0x08 jsr  $26,($27),0`	`R_ALPHA_LITUSE`	`4`
`0x0c ldah $29,0($26)`	`R_ALPHA_GPDISP`	`4`
`0x10 lda  $29,0($29)`		
⇓	⇓	⇓
`0x00 ldq  $16,x($gp)`	`R_ALPHA_GOTTPREL`	`x`
`0x04 unop`		
`0x08 call_pal PAL_rduniq`		
`0x0c addq $16,$0,$0`		
`0x10 unop`		
	Outstanding Relocations	
`GOT[n]`	`R_ALPHA_TPOFF64`	`x`

General Dynamic To Local Exec

GD ⟶ LE Code Transition	Initial Relocation	Symbol
`0x00 lda  $16,x($gp)`	`R_ALPHA_TLSGD`	`x`
`0x04 ldq  $27,__tls_get_addr($gp)`	`R_ALPHA_LITERAL`	`__tls_get_addr`
`0x08 jsr  $26,($27),0`	`R_ALPHA_LITUSE`	`4`
`0x0c ldah $29,0($26)`	`R_ALPHA_GPDISP`	`4`
`0x10 lda  $29,0($29)`		
⇓	⇓	⇓
`0x00 ldah $16,x($31)`	`R_ALPHA_TPRELHI`	`x`
`0x04 lda  $16,x($16)`	`R_ALPHA_TPRELLO`	`x`
`0x08 call_pal PAL_rduniq`		
`0x0c addq $16,$0,$0`		
`0x10 unop`		
	Outstanding Relocations	

This transition is used if the offset of `x` in the TLS block is within 2GiB. If the offset is larger, then the **GD** ⟶ **IE** transition is used, except that there is no dynamic relocation.

If the offset of `x` in the TLS block is within 32KiB, then the first instruction is an `lda` and the second instruction is a `unop`.

Local Dynamic To Local Exec

The **LD** ⟶ **LE** transitions are identical to the **GD** ⟶ **LE** transitions, except that we reference the base of the module's TLS section rather than a specific variable.

Initial Exec To Local Exec

IE ⟶ LE Code Transition	**Initial Relocation**	**Symbol**

`0x00 ldq  $1,x($gp)`
`0x04 addq $tp,$1,$0`
	R_ALPHA_GOTTPREL	x

⇓	⇓	⇓

`0x00 lda  $16,x($31)`
`0x04 addq $tp,$1,$0`
	R_ALPHA_TPREL	x

	Outstanding Relocations	

This transition is only used if the offset of `x` in the TLS block is within 32KiB. If the offset is larger, then the code sequence is unchanged, but the dynamic relocation in the GOT is removed.

5.5 x86-64 Linker Optimizations

x86-64 linker optimizations closely match IA-32 optimizations of GNU variants.

General Dynamic To Initial Exec

This code transition should explain the 4 byte padding in the general dynamic code sequence on x86-64. The IE sequence is 4 bytes longer:

GD ⟶ IE Code Transition	**Initial Relocation**	**Symbol**
`0x00 .byte 0x66`		
`0x01 leaq  x@tlsgd(%rip),%rdi`	R_X86_64_TLSGD	x
`0x08 .word 0x6666`		
`0x0a rex64`		
`0x0b call  __tls_get_addr@plt`	R_X86_64_PLT32	__tls_get_addr
⇓	⇓	⇓
`0x00 movq %fs:0,%rax`		
`0x09 addq x@gottpoff(%rip),%rax`	R_X86_64_GOTTPOFF	x
	Outstanding Relocations	
`GOT[n]`	R_X86_64_TPOFF64	x

General Dynamic To Local Exec

This transition is similar to the previous one, just the offset can be stored directly into the instruction:

GD ⟶ LE Code Transition	Initial Relocation	Symbol
`0x00 .byte 0x66`		
`0x01 leaq  x@tlsgd(%rip),%rdi`	R_X86_64_TLSGD	x
`0x08 .word 0x6666`		
`0x0a rex64`		
`0x0b call  __tls_get_addr@plt`	R_X86_64_PLT32	__tls_get_addr
⇓	⇓	⇓
`0x00 movq %fs:0,%rax`		
`0x09 leaq x@tpoff(%rax),%rax`	R_X86_64_TPOFF32	x
	Outstanding Relocations	

Local Dynamic to Local Exec

The following code transition requires padding in the resulting instruction:

LD ⟶ LE Code Transition	Initial Relocation	Symbol
`0x00 leaq x1@tlsld(%rip),%rdi`	R_X86_64_TLSLD	x1
`0x07 call __tls_get_addr@plt`	R_X86_64_PLT32	__tls_get_addr
`     ...`		
`0x10 leaq x1@dtpoff(%rax),%rcx`	R_X86_64_DTPOFF32	x1
⇓	⇓	⇓
`0x00 .word 0x6666`		
`0x02 .byte 0x66`		
`0x03 movq  %fs:0,%rax`		
`     ...`		
`0x10 leaq  x1@tpoff(%rax),%rdx`	R_X86_64_TPOFF32	x1
	Outstanding Relocations	

Initial Exec To Local Exec

The last of the x86-64 code transitions:

IE ⟶ LE Code Transition	**Initial Relocation**	**Symbol**
`0x00 movq %fs:0,%rax`		
`0x09 addq x@gottpoff(%rip),%rax`	R_X86_64_GOTTPOFF	x
⇓	⇓	⇓
`0x00 movq %fs:0,%rax`		
`0x09 leaq x@tpoff(%rax),%rax`	R_X86_64_TPOFF32	x
	Outstanding Relocations	

5.6 s390 Linker Optimizations

The s390 ABI defines the same four linker optimizations as IA-32. The optimizations explain the `__tls_get_offset` function. All code sequences for s390 consist of basically three things:

1. extract the thread pointer,
2. get the offset of the requested variable to the thread pointer, and
3. an operation on the variable with an index/base operand that combines the thread pointer and the offset (e.g. `la %rx,0(%ry,%rz)`).

All the optimizations have to do is to change the method how the offset is acquired.

General Dynamic To Initial Exec

The general dynamic access model is the most expensive one which makes this transition the most important one. For the general dynamic access model the code has to load a GOT offset from the literal pool and then call `__tls_get_offset` to get back the offset of the variable from the thread pointer. For the initial exec access model the code has to load a GOT entry that contains the offset of the variable from the thread pointer. One of the initial exec code variants uses a literal pool entry for the GOT offset. This makes the transition simple, the function call instruction is replaced by a load instruction and the literal pool constant `x@tlsgd` is replaced with `x@gotntpoff`:

GD ⟶ IE Code Transition	**Initial Relocation**	**Symbol**
`      l   %r6,.L1-.L0(%r13)`		
`      ear %r7,%a0`		
`      l   %r2,.L2-.L0(%r13)`		
`      bas %r14,0(%r6,%r13)`	R_390_TLS_GDCALL	x
`      la  %r8,0(%r2,%r7) # %r8 = &x`		
`      ...`		
`.L0: # literal pool, address in %r13`		
`.L1: .long __tls_get_offset@plt-.L0`		
`.L2: .long x@tlsgd`	R_390_TLS_GD32	x

⇓	⇓	⇓
`l   %r6,.L1-.L0(%r13)`		
`ear %r7,%a0`		
`l   %r2,.L2-.L0(%r13)`		
`l   %r2,0(%r2,%r12)`	R_390_TLS_LOAD	x
`la  %r8,0(%r2,%r7) # %r8 = &x`		
`...`		
`.L0: # literal pool, address in %r13`		
`.L1: .long __tls_get_offset@plt-.L0`		
`.L2: .long x@gotntpoff`	R_390_TLS_GOTIE32	x
	Outstanding Relocations	
`GOT[n]`	R_390_TLS_TPOFF32	x

General Dynamic To Local Exec

The optimization that turns the general dynamic code sequence into the local exec code sequence is as simple as the general dynamic to initial exec transition. The local exec code sequence loads the offset of the variable to the thread pointer directly from the literal pool. The function call instruction of the general dynamic code sequence is turned into a nop and the literal pool constant `x@tlsgd` is replaced with `x@ntpoff`:

GD $\longrightarrow$ LE Code Transition	**Initial Relocation**	**Symbol**
`l   %r6,.L1-.L0(%r13)`		
`ear %r7,%a0`		
`l   %r2,.L2-.L0(%r13)`		
`bas %r14,0(%r6,%r13)`	R_390_TLS_GDCALL	x
`la  %r8,0(%r2,%r7) # %r8 = &x`		
`...`		
`.L0: # literal pool, address in %r13`		
`.L1: .long __tls_get_offset@plt-.L0`		
`.L2: .long x@tlsgd`	R_390_TLS_GD32	x
⇓	⇓	⇓
`l   %r6,.L1-.L0(%r13)`		
`ear %r7,%a0`		
`l   %r2,.L2-.L0(%r13)`		
`bc  0,0 # nop`		
`la  %r8,0(%r2,%r7) # %r8 = &x`		
`...`		
`.L0: # literal pool, address in %r13`		
`.L1: .long __tls_get_offset@plt-.L0`		
`.L2: .long x@ntpoff`	R_390_TLS_LE32	x
	Outstanding Relocations	

Local Dynamic To Local Exec

The local dynamic to local exec code transition is a bit more complicated. To get the address of a thread local variable in the local dynamic model three things need to be added: the thread pointer, the (negative) offset to the TLS block of the module the code is in and the offset to the variable in the TLS block. The local exec code just has to add the thread pointer to the (negative) offset to the variable from the thread pointer. The transition is done be replacing the function call with a nop, the literal pool constant `x1@tlsldm` with 0 and the `@dtpoff` constants with `@ntpoff`:

LD → LE Code Transition	Initial Relocation	Symbol
`      l   %r6,.L1-.L0(%r13)`		
`      ear %r7,%a0`		
`      l   %r2,.L2-.L0(%r13)`		
`      bas %r14,0(%r6,%r13)`	`R_390_TLS_LDCALL`	`x1`
`      la  %r8,0(%r2,%r7)`		
`      l   %r9,.L3-.L0(%r13)`		
`      la  %r10,0(%r10,%r8) # %r10 = &x1`		
`      l   %r9,.L4-.L0(%r13)`		
`      la  %r10,0(%r10,%r8) # %r10 = &x2`		
`      ...`		
`.L0: # literal pool, address in %r13`		
`.L1: .long __tls_get_offset@plt-.L0`		
`.L2: .long x1@tlsldm`	`R_390_TLS_LDM32`	`x1`
`.L3: .long x1@dtpoff`	`R_390_TLS_LDO32`	`x1`
`.L4: .long x2@dtpoff`	`R_390_TLS_LDO32`	`x2`
⇓	⇓	⇓
`      l   %r6,.L1-.L0(%r13)`		
`      ear %r7,%a0`		
`      l   %r2,.L2-.L0(%r13)`		
`      bc  0,0 # nop`		
`      la  %r8,0(%r2,%r7)`		
`      l   %r9,.L3-.L0(%r13)`		
`      la  %r10,0(%r10,%r8) # %r10 = &x1`		
`      l   %r9,.L4-.L0(%r13)`		
`      la  %r10,0(%r10,%r8) # %r10 = &x2`		
`      ...`		
`.L0: # literal pool, address in %r13`		
`.L1: .long __tls_get_offset@plt-.L0`		
`.L2: .long 0`		
`.L3: .long x1@ntpoff`	`R_390_TLS_LE32`	`x1`
`.L4: .long x2@ntpoff`	`R_390_TLS_LE32`	`x2`

	Outstanding Relocations	
`GOT[n]`	`R_390_TLS_DTPMOD`	`x1`

Initial Exec To Local Exec

The code transition from initial exec to local exec doesn't improve the execution speed but for two of the three initial exec variants a GOT entry less is needed.

IE ⟶ LE Code Transition	**Initial Relocation**	**Symbol**
`      ear %r7,%a0`		
`      l   %r8,.L1-.L0(%r13)`		
`      l   %r9,0(%r8,%r12)`	`R_390_TLS_LOAD`	`x`
`      la  %r10,0(%r9,%r7) # %r10 = &x`		
`      ...`		
`.L0: # literal pool, address in %r13`		
`.L1: .long x@gotntpoff`	`R_390_TLS_GOTIE32`	`x`
⇓	⇓	⇓
`      ear %r7,%a0`		
`      l   %r8,.L1-.L0(%r13)`		
`      lr  %r9,%r8 ; bcr 0,%r0`		
`      la  %r10,0(%r9,%r7) # %r10 = &x`		
`      ...`		
`.L0: # literal pool, address in %r13`		
`.L1: .long x@ntpoff`	`R_390_TLS_LE32`	`x`
	Outstanding Relocations	

IE ⟶ LE Code Transition	**Initial Relocation**	**Symbol**
`      ear %r7,%a0`		
`      l   %r8,.L1-.L0(%r13)`		
`      l   %r9,0(%r8)`	`R_390_TLS_LOAD`	`x`
`      la  %r10,0(%r9,%r7) # %r10 = &x`		
`      ...`		
`.L0: # literal pool, address in %r13`		
`.L1: .long x@indntpoff`	`R_390_TLS_GOTIE32`	`x`
⇓	⇓	⇓
`      ear %r7,%a0`		
`      l   %r8,.L1-.L0(%r13)`		
`      lr  %r9,%r8 ; bcr 0,%r0`		

`      la  %r10,0(%r9,%r7) # %r10 = &x`		
`      ...`		
`.L0: # literal pool, address in %r13`		
`.L1: .long x@ntpoff`	R_390_TLS_LE32	x
	Outstanding Relocations	

There is no IE ⟶ LE code transition for the small GOT case because no literal pool entry exists where the modified constant `x@ntpoff` could be stored. For this case a slot in the GOT is used for the constant.

5.7 s390x Linker Optimizations

The same four optimizations as for s390 are available for s390x. The optimizations follow the same principles but with 64 bit instructions instead of 32 bit instructions. The 6 byte `brasl` instruction is replaced with either the 6 byte `lg` load instruction or the 6 byte `brcl 0,.` nop. The 6 byte `lg` instruction is replaced with the 6 byte triadic shift by 0 bit `sllg` that is used instead of the more appropriate `lgr` which unfortunatly has only 4 byte.

General Dynamic to Initial Exec

GD ⟶ IE Code Transition	**Initial Relocation**	**Symbol**
`      ear   %r7,%a0`		
`      sllg  %r7,%r7,32`		
`      ear   %r7,%a1`		
`      lg    %r2,.L1-.L0(%r13)`		
`      brasl %r14,__tls_get_offset@plt`	R_390_TLS_GDCALL	x
`      la    %r8,0(%r2,%r7) # %r8 = &x`		
`      ...`		
`.L0: # literal pool, address in %r13`		
`.L1: .quad x@tlsgd`	R_390_TLS_GD64	x
⇓	⇓	⇓
`      ear   %r7,%a0`		
`      sllg  %r7,%r7,32`		
`      ear   %r7,%a1`		
`      lg    %r2,.L1-.L0(%r13)`		
`      lg    %r2,0(%r2,%r12)`	R_390_TLS_LOAD	x
`      la    %r8,0(%r2,%r7) # %r8 = &x`		
`      ...`		
`.L0: # literal pool, address in %r13`		
`.L1: .quad x@gotntpoff`	R_390_TLS_GOTIE64	x
	Outstanding Relocations	
`GOT[n]`	R_390_TLS_TPOFF64	x

General Dynamic to Local Exec

GD ⟶ LE Code Transition	Initial Relocation	Symbol
`      ear   %r7,%a0`		
`      sllg  %r7,%r7,32`		
`      ear   %r7,%a1`		
`      lg    %r2,.L1-.L0(%r13)`		
`      brasl %r14,__tls_get_offset@plt`	`R_390_TLS_GDCALL`	`x`
`      la    %r8,0(%r2,%r7) # %r8 = &x`		
`      ...`		
`.L0: # literal pool, address in %r13`		
`.L1: .quad x@tlsgd`	`R_390_TLS_GD64`	`x`
⇓	⇓	⇓
`      ear   %r7,%a0`		
`      sllg  %r7,%r7,32`		
`      ear   %r7,%a1`		
`      lg    %r2,.L1-.L0(%r13)`		
`      brcl  0,.`		
`      la    %r8,0(%r2,%r7) # %r8 = &x`		
`      ...`		
`.L0: # literal pool, address in %r13`		
`.L1: .quad x@ntpoff`	`R_390_TLS_LE64`	`x`
	Outstanding Relocations	

Local Dynamic to Local Exec

LD ⟶ LE Code Transition	Initial Relocation	Symbol
`      ear   %r7,%a0`		
`      sllg  %r7,%r7,32`		
`      ear   %r7,%a1`		
`      lg    %r2,.L1-.L0(%r13)`		
`      brasl %r14,__tls_get_offset@plt`	`R_390_TLS_LDCALL`	`x1`
`      la    %r8,0(%r2,%r7)`		
`      lg    %r9,.L2-.L0(%r13)`		
`      la    %r10,0(%r9,%r8) # %r10 = &x1`		
`      lg    %r9,.L3-.L0(%r13)`		
`      la    %r10,0(%r9,%r8) # %r10 = &x2`		
`      ...`		
`.L0: # literal pool, address in %r13`		
`.L1: .quad x1@tlsldm`	`R_390_TLS_LDM64`	`x1`

`.L2: .quad x1@dtpoff`	R_390_TLS_LDO64	x1
`.L3: .quad x2@dtpoff`	R_390_TLS_LDO64	x2
⇓	⇓	⇓
`      ear   %r7,%a0`		
`      sllg  %r7,%r7,32`		
`      ear   %r7,%a1`		
`      lg    %r2,.L1-.L0(%r13)`		
`      brcl  0,.`		
`      la    %r8,0(%r2,%r7)`		
`      lg    %r9,.L2-.L0(%r13)`		
`      la    %r10,0(%r9,%r8) # %r10 = &x1`		
`      lg    %r9,.L3-.L0(%r13)`		
`      la    %r10,0(%r9,%r8) # %r10 = &x2`		
`      ...`		
`.L0: # literal pool, address in %r13`		
`.L1: .quad 0`		
`.L2: .quad x1@ntpoff`	R_390_TLS_LE64	x1
`.L3: .quad x2@ntpoff`	R_390_TLS_LE64	x2
	Outstanding Relocations	

Initial Exec to Local Exec

IE ⟶ LE Code Transition	**Initial Relocation**	**Symbol**
`      ear  %r7,%a0`		
`      sllg %r7,%r7,32`		
`      ear  %r7,%a1`		
`      lg   %r8,.L1-.L0(%r13)`		
`      lg   %r9,0(%r8,%r12)`	R_390_TLS_LOAD	x
`      la   %r10,0(%r9,%r7) # %r10 = &x`		
`      ...`		
`.L0: # literal pool, address in %r13`		
`.L1: .quad x@gotntpoff`	R_390_TLS_GOTIE64	x
⇓	⇓	⇓
`      ear  %r7,%a0`		
`      sllg %r7,%r7,32`		
`      ear  %r7,%a1`		
`      lg   %r8,.L1-.L0(%r13)`		
`      sllg %r9,%r8,0`		
`      la   %r10,0(%r9,%r7) # %r10 = &x`		
`      ...`		
`.L0: # literal pool, address in %r13`		
`.L1: .quad x@ntpoff`	R_390_TLS_LE64	x
	Outstanding Relocations	

IE → LE Code Transition	**Initial Relocation**	**Symbol**
`      ear  %r7,%a0`		
`      sllg %r7,%r7,32`		
`      ear  %r7,%a1`		
`      lg   %r8,.L1-.L0(%r13)`		
`      lg   %r9,0(%r8)`	`R_390_TLS_LOAD`	`x`
`      la   %r10,0(%r9,%r7) # %r10 = &x`		
`      ...`		
`.L0: # literal pool, address in %r13`		
`.L1: .quad x@indntpoff`	`R_390_TLS_GOTIE64`	`x`
⇓	⇓	⇓
`      ear  %r7,%a0`		
`      sllg %r7,%r7,32`		
`      ear  %r7,%a1`		
`      lg   %r8,.L1-.L0(%r13)`		
`      sllg %r9,%r8,0`		
`      la   %r10,0(%r9,%r7) # %r10 = &x`		
`      ...`		
`.L0: # literal pool, address in %r13`		
`.L1: .quad x@ntpoff`	`R_390_TLS_LE64`	`x`
	Outstanding Relocations	

6 New ELF Definitions

This section shows the actual definitions for the newly introduced constants necessary to describe the extended ELF format. The generic extensions are:

```
#define SHF_TLS   (1 << 10)

#define STT_TLS   6

#define PT_TLS    7
```

6.1 New IA-64 ELF Definitions

```
#define R_IA64_TPREL14        0x91 /* @tprel(sym+add),imm14 */
#define R_IA64_TPREL22        0x92 /* @tprel(sym+add),imm22 */
#define R_IA64_TPREL64I       0x93 /* @tprel(sym+add),imm64 */
#define R_IA64_TPREL64MSB     0x96 /* @tprel(sym+add),data8 MSB */
#define R_IA64_TPREL64LSB     0x97 /* @tprel(sym+add),data8 LSB */
#define R_IA64_LTOFF_TPREL22  0x9a /* @ltoff(@tprel(s+a)),imm2 */
#define R_IA64_DTPMOD64MSB    0xa6 /* @dtpmod(sym+add),data8 MSB */
#define R_IA64_DTPMOD64LSB    0xa7 /* @dtpmod(sym+add),data8 LSB */
#define R_IA64_LTOFF_DTPMOD22 0xaa /* @ltoff(@dtpmod(sym+add)),imm22 */
#define R_IA64_DTPREL14       0xb1 /* @dtprel(sym+add),imm14 */
#define R_IA64_DTPREL22       0xb2 /* @dtprel(sym+add),imm22 */
#define R_IA64_DTPREL64I      0xb3 /* @dtprel(sym+add),imm64 */
#define R_IA64_DTPREL32MSB    0xb4 /* @dtprel(sym+add),data4 MSB */
#define R_IA64_DTPREL32LSB    0xb5 /* @dtprel(sym+add),data4 LSB */
#define R_IA64_DTPREL64MSB    0xb6 /* @dtprel(sym+add),data8 MSB */
#define R_IA64_DTPREL64LSB    0xb7 /* @dtprel(sym+add),data8 LSB */
#define R_IA64_LTOFF_DTPREL22 0xba /* @ltoff(@dtprel(s+a)), imm22 */
```

The operators used in the code sequences are defined as follows:

`@ltoff(`*expr*`)` Requests the creation of a GOT entry that will hold the full value of *expr* and computes the `gp`-relative displacement to that GOT entry.

`@tprel(`*expr*`)` Computes a `tp`-relative displacement – the difference between the effective address and the value of the thread pointer. The expression must evaluate to an effective address within a thread-specific data segment.

`@dtpmod(`*expr*`)` Computes the load module index corresponding to the load module that contains the definition of the symbol referenced by the relocation. When used in conjunction with the `@ltoff()` operator, it evaluates to the `gp`-relative offset of a linkage table entry that holds the computed load module index.

`@dtprel(`*expr*`)` Computes a dtv-relative displacement – the difference between the effective address and the base address of the thread-local storage block that contains the definition of the symbol referenced by the relocation. When used in conjunction with the `@ltoff()` operator, it evaluates to the `gp`-relative offset of a linkage table entry that holds the computed displacement.

6.2 New IA-32 ELF Definitions

```
#define R_386_TLS_TPOFF     14 /* Negative offset in static TLS
                                  block (GNU version) */
#define R_386_TLS_IE        15 /* Absolute address of GOT entry
                                  for negative static TLS block
                                  offset */
#define R_386_TLS_GOTIE     16 /* GOT entry for negative static
                                  TLS block offset */
#define R_386_TLS_LE        17 /* Negative offset relative to
                                  static TLS (GNU version) */
#define R_386_TLS_GD        18 /* Direct 32 bit for GNU version
                                  of GD TLS */
#define R_386_TLS_LDM       19 /* Direct 32 bit for GNU version
                                  of LD TLS in LE code */
#define R_386_TLS_GD_32     24 /* Direct 32 bit for GD TLS */
#define R_386_TLS_GD_PUSH   25 /* Tag for pushl in GD TLS
                                  code */
#define R_386_TLS_GD_CALL   26 /* Relocation for call to
#define R_386_TLS_GD_POP    27 /* Tag for popl in GD TLS
                                  code */
#define R_386_TLS_LDM_32    28 /* Direct 32 bit for local
                                  dynamic code */
#define R_386_TLS_LDM_PUSH  29 /* Tag for pushl in LDM TLS
                                  code */
#define R_386_TLS_LDM_CALL  30 /* Relocation for call to
#define R_386_TLS_LDM_POP   31 /* Tag for popl in LDM TLS
                                  code */
#define R_386_TLS_LDO_32    32 /* Offset relative to TLS
                                  block */
#define R_386_TLS_IE_32     33 /* GOT entry for static TLS
                                  block */
#define R_386_TLS_LE_32     34 /* Offset relative to static
                                  TLS block */
#define R_386_TLS_DTPMOD32  35 /* ID of module containing
                                  symbol */
#define R_386_TLS_DTPOFF32  36 /* Offset in TLS block */
#define R_386_TLS_TPOFF32   37 /* Offset in static TLS
                                  block */
```

The operators used in the code sequences are defined as follows:

`@dtlndx(`*expr*`)` Allocate two contiguous entries in the GOT to hold a `tls_index` structure (for passing to `__tls_get_addr`). The instructions referencing this entry will be bound to the first of the two GOT entries.

`@tlsgd(`*expr*`)` This expression is the eqpuivalent of `@dtlndx(`*expr*`)` for the GNU variant of the calling conventions. The linker is also allocating to two consecutive entries in the GOT and the processing of the relocation produces the offset of the first entry as the value of the expression. The only difference is that the function called is `___tls_get_addr`.

`@tlsplt` This expression is used in the `call` instructions to `__tls_get_addr` in the Sun variants. The `call` instruction is associated with the symbol the whole code sequence this instruction is part of deals with. The `R_386_TLS_`*xxx*`_CALL` relocations generated for the `call` instructions will reference the symbol. The linker will insert a reference to `__tls_get_addr`.

`@tmndx(`*expr*`)` Allocate two contiguous entries in the GOT to hold a `tls_index` structure (for passing to `__tls_get_addr`). The `ti_offset` field of the object will be set to 0 (zero) and the `ti_module` field will be filled in at run-time. The call to `__tls_get_addr` will return the starting offset of the dynamic TLS block.

`@tlsldm(`*expr*`)` This expression is the GNU variant of `@tmndx(`*expr*`)`. Just as for `@tlsgd(`*expr*`)` the only difference is that the function called in the following `call` instruction is `___tls_get_addr`.

`@dtpoff` Calculate the offset of the variable it is added to relative to the TLS block it is contained in. The value is used as an immediate value of an addend and is not associated with a specific register.

`@tpoff` Calculate the offset of the variable it is added to relative to the static TLS block. The linker allocates one GOT entry for the result of the relocation.

The operator must be used to compute an immediate value. The linker will report an error if the referenced variable is not defined or it is not code for the executable itself. No GOT entry is created in this case.

If used in the form `@tpoff(`*expr*`)` the offset of the variable in *expr* relative to the static TLS block is calculated. The linker allocates one GOT entry for the result of the relocation.

`@ntpoff` Calculate the negative offset of the variable it is added to relative to the static TLS block.

The operator must be used to compute an immediate value. The linker will report an error if the referenced variable is not defined or it is not code for the executable itself. No GOT entry is created in this case.

`@gotntpoff` This expression is the GNU variant of `@tpoff(`*expr*`)`. The difference is that the GOT slot allocated by it it must be added to a variable and that the relocation is for both `movl` and `addl` assembler instruction which is relevant of the code sequence is transformed to the Local Exec model by the linker.

The `@gotntpoff` is also **not** used for immediate instructions. Instead the GNU variant of the Local Exec model will also use the `@tpoff` expression. Since the Local Exec model is as simple as it gets the linker does not have to be aware of the differences of the two variants. No conversion can be performed and therefore the expression is used exclusively to get the linker fill in the correct offset.

`@indntpoff` This expression is similar to `@gotntpoff`, but for use in position dependent code. While `@gotntpoff` resolves to GOT slot address relative to the start of the GOT in the `movl` or `addl` instructions, `@indntpoff` resolves to the absolute GOT slot address.

6.3 New SPARC ELF Definitions

```
#define R_SPARC_TLS_GD_HI22     56
#define R_SPARC_TLS_GD_LO10     57
#define R_SPARC_TLS_GD_ADD      58
#define R_SPARC_TLS_GD_CALL     59
#define R_SPARC_TLS_LDM_HI22    60
#define R_SPARC_TLS_LDM_LO10    61
#define R_SPARC_TLS_LDM_ADD     62
#define R_SPARC_TLS_LDM_CALL    63
#define R_SPARC_TLS_LDO_HIX22   64
#define R_SPARC_TLS_LDO_LOX10   65
#define R_SPARC_TLS_LDO_ADD     66
#define R_SPARC_TLS_IE_HI22     67
#define R_SPARC_TLS_IE_LO10     68
#define R_SPARC_TLS_IE_LD       69
#define R_SPARC_TLS_IE_LDX      70
#define R_SPARC_TLS_IE_ADD      71
#define R_SPARC_TLS_LE_HIX22    72
#define R_SPARC_TLS_LE_LOX10    73
#define R_SPARC_TLS_DTPMOD32    74
#define R_SPARC_TLS_DTPMOD64    75
#define R_SPARC_TLS_DTPOFF32    76
#define R_SPARC_TLS_DTPOFF64    77
#define R_SPARC_TLS_TPOFF32     78
#define R_SPARC_TLS_TPOFF64     79
```

The operators used in the code sequences are defined as follows:

`@dtlndx(`*expr*`)` Allocate two contiguous entries in the GOT to hold a `tls_index` structure (for passing to `__tls_get_addr`). The instructions referencing this entry will be bound to the first of the two GOT entries.

`@tmndx(`*expr*`)` Allocate two contiguous entries in the GOT to hold a `tls_index` structure (for passing to `__tls_get_addr`). The `ti_offset` field of the object will be set to 0 (zero) and the `ti_module` field will be filled in at run-time. The call to `__tls_get_addr` will return the starting offset of the dynamic TLS block.

`@dtpoff(`*expr*`)` Calculate the offset of the variable in *expr* relative to the TLS block it is contained in.

`@tpoff(`*expr*`)` Calculate the negative offset of the variable in *expr* relative to the static TLS block.

6.4 New SH ELF Definitions

```
#define R_SH_TLS_GD_32 144
#define R_SH_TLS_LD_32 145
#define R_SH_TLS_LDO_32 146
#define R_SH_TLS_IE_32 147
```

```
#define R_SH_TLS_LE_32 148
#define R_SH_TLS_DTPMOD32 149
#define R_SH_TLS_DTPOFF32 150
#define R_SH_TLS_TPOFF32 151
```

The operators used in the code sequences are defined as follows:

`@tlsgd(`*expr*`)` This expression is the eqpuivalent of `@dtlndx(`*expr*`)` for the GNU variant of the calling conventions. The linker is also allocating to two consecutive entries in the GOT and the processing of the relocation produces the offset of the first entry as the value of the expression. The only difference is that the function called is `___tls_get_addr`.

`@tlsldm(`*expr*`)` This expression is the GNU variant of `@tmndx(`*expr*`)`. Just as for `@tlsgd(`*expr*`)` the only difference is that the function called in the following `call` instruction is `___tls_get_addr`.

`@dtpoff` Calculate the offset of the variable it is added to relative to the TLS block it is contained in. The value is used as an immediate value of an addend and is not associated with a specific register.

`@tpoff` Calculate the offset of the variable it is added to relative to the static TLS block. The linker allocates one GOT entry for the result of the relocation.

The operator must be used to compute an immediate value. The linker will report an error if the referenced variable is not defined or it is not code for the executable itself. No GOT entry is created in this case.

If used in the form `@tpoff(`*expr*`)` the offset of the variable in *expr* relative to the static TLS block is calculated. The linker allocates one GOT entry for the result of the relocation.

`@gottpoff` Represents the offset in the GOT for the entry which contains the `tls_index` entries for the variable the relocation is attached to.

6.5 New x86-64 ELF Definitions

```
#define R_X86_64_DTPMOD64   16 /* ID of module containing
                                  symbol */
#define R_X86_64_DTPOFF64   17 /* Offset in TLS block */
#define R_X86_64_TPOFF64    18 /* Offset in initial TLS
                                  block */
#define R_X86_64_TLSGD      19 /* PC relative offset to GD GOT
                                  block */
#define R_X86_64_TLSLD      20 /* PC relative offset to LD GOT
                                  block */
#define R_X86_64_DTPOFF32   21 /* Offset in TLS block */
#define R_X86_64_GOTTPOFF   22 /* PC relative offset to IE GOT
                                  entry */
#define R_X86_64_TPOFF32    23 /* Offset in initial TLS
                                  block */
```

The operators used in the code sequences are defined as follows:

`@tlsgd(%rip)` Allocate two contiguous entries in the GOT to hold a `tls_index` structure (for passing to `__tls_get_addr`). It may be used only in the exact x86-64 general dynamic code sequence shown above.

`@tlsld(%rip)` Allocate two contiguous entries in the GOT to hold a `tls_index` structure (for passing to `__tls_get_addr`). The `ti_offset` field of the object will be set to 0 (zero) and the `ti_module` field will be filled in at run-time. The call to `__tls_get_addr` will return the starting offset of the dynamic TLS block. It may be only used in the exact code sequence as shown above.

`@dtpoff` Calculate the offset of the variable relative to the start of the TLS block it is contained in. The value is used as an immediate value of an addend and is not associated with a specific register.

`@gottpoff(%rip)` Allocate one GOT entry to hold a variable offset in initial TLS block (relative to TLS block end, `%fs:0`). The operator must be used in `movq` or `addq` instructions only.

`@tpoff` Calculate the offset of the variable relative to TLS block end, `%fs:0`. No GOT entry is created.

6.6 New s390/s390x ELF Definitions

```
#define R_390_TLS_LOAD    37 /* Tag for load insn in TLS code */
#define R_390_TLS_GDCALL  38 /* Tag for call insn in TLS code */
#define R_390_TLS_LDCALL  39 /* Tag for call insn in TLS code */
#define R_390_TLS_GD32    40 /* Direct 32 bit for general dynamic
                                thread local data */
#define R_390_TLS_GD64    41 /* Direct 64 bit for general dynamic
                                thread local data */
#define R_390_TLS_GOTIE12 42 /* 12 bit GOT offset for static TLS
                                block offset */
#define R_390_TLS_GOTIE32 43 /* 32 bit GOT offset for static TLS
                                block offset */
#define R_390_TLS_GOTIE64 44 /* 64 bit GOT offset for static TLS
                                block offset*/
#define R_390_TLS_LDM32   45 /* Direct 32 bit for local dynamic
                                thread local data in LE code */
#define R_390_TLS_LDM64   46 /* Direct 64 bit for local dynamic
                                thread local data in LE code */
#define R_390_TLS_IE32    47 /* 32 bit address of GOT entry for
                                negated static TLS block offset */
#define R_390_TLS_IE64    48 /* 64 bit address of GOT entry for
                                negated static TLS block offset */
#define R_390_TLS_IEENT   49 /* 32 bit rel. offset to GOT entry for
```

```
                                 negated static TLS block offset */
#define R_390_TLS_LE32    50 /* 32 bit negated offset relative
                                 to static TLS block */
#define R_390_TLS_LE64    51 /* 64 bit negated offset relative
                                 to static TLS block */
#define R_390_TLS_LDO32   52 /* 32 bit offset relative to TLS
                                 block */
#define R_390_TLS_LDO64   53 /* 64 bit offset relative to TLS
                                 block */
#define R_390_TLS_DTPMOD  54 /* ID of module containing symbol */
#define R_390_TLS_DTPOFF  55 /* Offset in TLS block */
#define R_390_TLS_TPOFF   56 /* Negated offset in static TLS
                                 block */
```

The operators used in the code sequences are defined as follows:

`@tlsgd` Allocate two contiguous entries in the GOT to hold a `tls_index` structure. The value of the expression `x@tlsgd` is the offset from the start of the GOT to the `tls_index` structure for the symbol `x`. The call to `__tls_get_offset` with the GOT offset to the `tls_index` structure of x will return the offset of the thread local variable x to the TCB pointer. The `@tlsgd` operator may be used only in the general dynamic access model as shown above.

`@tlsldm` Allocate two contiguous entries in the GOT to hold a `tls_index` structure. The `ti_offset` field of the object will be set to 0 (zero) and the `ti_module` field will be filled in a at run-time. The value of the expression `x@tlsldm` is the offset from the start of the GOT to this special `tls_index` structure. The call to `__tls_get_offset` with the GOT offset to this special `tls_index` structure will return the offset of the dynamic TLS block to the TCB pointer. The `@tlsgd` operator may be used only in the local dynamic access model as shown above.

`@dtpoff` Calculate the offset of the variable relative to the start of the TLS block it is contained in. The `@dtpoff` operator may be used only in the local dynamic access model as shown above.

`@ntpoff` The value of the expression `x@ntpoff` is the offset of the thread local variable `x` relative to the TCB pointer. No GOT entry is created in this case. The `@ntpoff` operator may be used only in the local exec model as shown above.

`@gotntpoff` Allocate a GOT entry to hold the offset of a variable in the initial TLS block relative to the TCB pointer. The value of of the expression `x@gotntpoff` is offset in the GOT to the allocated entry. The `@gotntpoff` operator may be used only in the initial exec model as shown above.

`@indntpoff` This expression is similar to `@gotntpoff`. The difference is that the value of `x@indntpoff` is not a GOT offset but the address of the allocated GOT entry itself. It is used in position dependent code and in combination with the `larl` instruction. The `@indntpoff` operator may be used only in the initial exec model as shown above.

7 Revision History

2002-1-27 First version. The information and structure of the document is mainly based on the IA-64 ABI document and the document from Sun Microsystems (written by Mike Walker <`Michael.Walker@sun.com`>) describing their implementation for SPARC and IA-32.

2002-1-31 More edits. Fix some embarrassing mistakes. Better wording.

2002-2-17 Update GNU-specific i386 definitions. Internal changes to use more standard macro packages.

2002-4-8 Integrated the SH ABI. Based on the notes from Kaz Kojima <`kkojima@rr.iij4u.or.jp`>.

2002-5-20 Update GNU x86 code sequences. Mostly removal of `nop`s. Introduce `@ntpoff` and its relocation.

2002-9-18 Update GNU x86 code sequences. Introduce `@gotntpoff`, `@indntpoff` and its relocations.

2002-9-23 Some typos and erros fixed by Andreas Jaeger <`aj@suse.de`>.

2002-10-15 Complete SH ABI.

2002-10-21 x86-64 ABI description added by Jakub Jelínek <`jakub@redhat.com`>.

2002-10-23 Alpha ABI added by Richard Henderson <`rth@redhat.com`>.

2002-10-23 Publish x86 code to directly access memory from via %gs.

2002-11-9 Internal TEX code improvements.

2002-11-14 Correct contradicting description of IA-32 `@ntpoff`.

2003-1-28 Integrate s390 description. Contributed by Martin Schwidefsky <`schwidefsky@de.ibm.com`>.