Speeding Up Thread-Local Storage Access in Dynamic Libraries in the ARM platform

Glauber de Oliveira Costa
IC-Unicamp
glommer@gmail.com

Alexandre Oliva
Red Hat and IC-Unicamp
aoliva@redhat.com, oliva@lsd.ic.unicamp.br

Abstract

As multi-core processors become the rule rather than the exception, multi-threaded programming is expected to expand from its current niches to more widespread use, in software components that have not traditionally been concerned about exploiting concurrency. Accessing thread-local storage (TLS) from within dynamic libraries has traditionally required calling a function to obtain the thread-local address of the variable. Such function calls are several times slower than typical addressing code that is used in executables. While instructions used in executables can assume thread-local variables are at a constant offset within the thread Static TLS block, dynamic libraries loaded during program execution may not even assume that their thread-local variables are in Static TLS blocks.

Since libraries are most commonly loaded as dependencies of executables or other libraries, before a program starts running, the most common TLS case is that of constant offsets. Recently, an access model that enables dynamic libraries to take advantage of this fact without giving up the ability to be loaded during execution was proposed and succesfully implemented on IA32, AMD64/EM64T and Fujitsu FR-V architectures. On these systems, experimental results revealed the new model consistently exceeds the old model in terms of performance, particularly in the most common case, where the speedup is often well over 2x, bringing it nearly to the same performance of access models used in plain executables.

This paper details this new access model and its implementation for ARM processors, highlighting its particular issues and potential gains in embedded systems.

1 Introduction

As mainstream microprocessor vendors turn to multi-core processors as a way to improve performance [1, 2], the relevance of multi-threaded programming to leverage on such potential performance improvements grows. Embedded systems pose their own challenges, such as delivering high performance with minimum size and low power consumption.

Besides the common difficulty multi-threaded programs run into, namely the need for synchronization between threads, it is often the case that a thread would like to use a global variable,[1] for extended periods of time, without other threads modifying its contents, and without having to incur synchronization overheads.

Using automatic variables to achieve this is a possibility, since each thread has its own stack, where such variables are allocated. However, if multiple functions need to use the same data structure within a thread, a pointer to it must be passed around, which is cumbersome, and might require re-engineering the control flow so as to ensure that the stack frame in which the data structure is created is not left while the data is still in use.

Widely-used thread libraries have introduced primitives to overcome this problem, enabling threads to map a global handle, shared by all threads, to different values, one for each thread. This feature is offered in the form of function calls (`pthread_getspecific` and `pthread_setspecific`, in POSIX [3] threads), that are far less efficient than access to global or automatic variables. Besides the efficiency issues, they are syntactically far more difficult to use than regular variables. These were the main motivations for the introduction of Thread Local Storage (henceforth, TLS[4, 5]) features in compilers, linkers and run-time systems, that enable selected global variables to be marked with a `__thread` specifier or a `threadprivate` pragma, indicating that, for each thread, there should be a separate, independent copy of the variable.

By using custom low-level thread-specific implementations [6], or with cooperation from the compiler and the linker, access to thread-local variables can be far more efficient than using the standard functions that offer abstractions of thread-specific data. In some cases, such as when generating code for dynamic libraries, the compiler-generated code is still very inefficient [4] ; for main executables, access can sometimes be just as efficient as accessing automatic or global variables. The mechanism proposed by Oliva and Araújo [4] yields a major speedup, that brings the performance of TLS access in dynamic libraries close to that of executables. Such a mechanism has been proved successful after implemented in the Fujitsu FR-V, x86_64 and i386 architectures. On the ARM platform, a typical choice for embedded systems [7], work has been done to provide such improvements, while keeping in mind all the restrictions that may be imposed upon embedded systems

[1]The strictly-correct term here would be variable whose storage has static duration.

1.1 Terminology and organization

In this paper, we use the term *loadable module*, or just *module*, to refer to executables, dynamic libraries and the dynamic loader. A process may consist of a set of loadable modules consisting of exactly one executable, a dynamic loader (for dynamic executables) and zero or more dynamic libraries. We call *initial modules* the main executable, any dynamic libraries it depends upon (directly or indirectly) and any other dynamic libraries the dynamic loader chooses to load before relinquishing control to the main executable. Moreover, we use the term *dlopened modules* to refer to modules that are loaded after the program starts running, typically by means of library calls such as `dlopen`.

This paper is organized as follows: section 2 gives background material about TLS symbols and the novel concept of TLS descriptors. Section 3 details the proposed extensions in the ABI for enabling it in the ARM platform, while section 4 unveils the needed changes in the current ABI and the tools covered by this work. Performance measures are then shown in section 5. Finally, section 6 sheds light on what there is yet to be done in the field of TLS access, specially for its broader acceptance.

2 Background

GCC, since version 3.3 [8], has provided the ability of marking a variable with a `__thread` modifier, which causes it to be marked as a TLS symbol. Each module containing TLS symbols has a section containing the initial values in the TLS block. For all TLS symbols this module exports, there is a fixed offset within such a block in which the symbol can be found.

An initial module is guaranteed to have its TLS data laid out as part of the Static TLS block, a per-thread data structure whose per-thread address is held in a thread pointer, normally a reserved register. Since the same layout is used for the Static TLS blocks of all threads, the relative address from the thread pointer to a symbol defined in such a module is constant for all threads.

At a fixed location in the Static TLS block, there is a pointer to another data structure called the Dynamic Thread Vector, henceforth, DTV. When a module is `dlopened` or `dlclosed` the thread's DTV may have to be modified to add or remove the module's TLS block.

Figure 1 shows these data structure and the iterations between them. Access to a TLS symbol can be performed in any one of four models:

Initial Exec is used when the symbol is in a module loaded at start-up time with the executable. The relative address is a fixed offset from the thread pointer, and can be written to a specific GOT entry at load time. The main drawback of this access model is that, by using it, we give up the ability to dynamically load the module.

Local Exec is used when the symbol is defined in the main executable. In such a case, no additional effort is needed to compute the variable address, that is at a constant offset from the thread pointer known at link time. The model is not suitable for symbols defined in a dynamic library, because even though the offset will be a constant at run time, it is not a link-time constant.

General Dynamic is the most general of all four models, covering both the initial and dynamic module load cases. Address resolution goes through a call to `__tls_get_addr`, that by a lookup in the DTV, loads the variable address. As parameters, it receives the module identifier and the symbol offset within the module's TLS section.

Local Dynamic is a variant of the General Dynamic model for symbols living in the same module. The call to `__tls_get_addr` receives a zero-offset parameter, in a way that it returns the base address of the module. Subsequent access may then just add to it the symbol offset.

In some platforms, when a module that has symbols using the initial exec or dynamic models is linked into an executable, the linker can perform a relaxation step, allowing the access to be turned into more efficient ones, without any loss of generality (dlopening executables does not make sense). In the ARM current ABI, no such relaxations are specified.

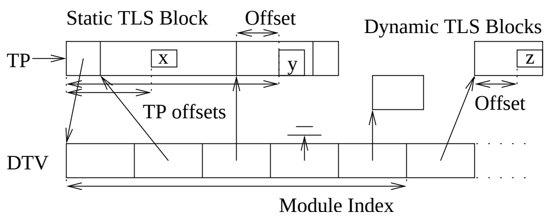

Figure 1: Data structures used for TLS handling. Static TLS block, the DTV, and the iterations between them

For dynamic libraries, whether the module is initially loaded with the executable or `dlopened`, access go through a call to `__tls_get_addr`, whose two arguments, the module ID and the symbol offset inside the module are stored in two GOT entries. However, for the most common case of a module being loaded along with the main executable, the variable address lies in the Static TLS block, therefore being at a constant offset from the thread pointer.

The main idea behind TLS descriptors [9] is to take advantage of this common case by using the same number of GOT entries the original TLS model uses, in a slightly different way. One of the entries is used by a load-time chosen specialization function, while the remaining space is filled with a parameter to this function, whose meaning is dependant of the type of specialization.

In the case the module is loaded along with the executable, we use the thread-pointer offset as the parameter, and the specialization call just returns it, without the need to issue a call to the time-consuming `__tls_get_addr`. For the dynamic load case, the specialization uses the information provided by the parameter to call `__tls_get_addr` or to perform equivalent optimized steps it may see fit. Other specializations can also be used for some more specific cases.

Ideally, we should be also able to relax the code sequences in case of linking into executables, which is missing in the current ABI.

3 ARM TLS Descriptors

The ARM ABI currently specifies the use of two consecutive GOT entries [10] for TLS symbols relocations. In the same amount of space, we store TLS descriptors. The first entry contains the parameter for the specialized function while the second holds the pointer to the function itself. This slight difference from the previous implementations [11], that uses the first parameter as the specialization while the second holds the parameter, allowed us to keep the calling convention of parameter passing primarily in the `r0` register, with no additional effort.

As we use new relocations to drive our mechanism, our model can also coexist with modules that uses the old TLS model, i.e. old modules can be linked with TLS Descriptors-enabled ones with no penalties other than a size increase of 8 bytes per symbol that uses both models, as each relocation will point to a different pair of GOT entries.

If the size increase is not worth the performance gain for some workload, it's still possible to go with the old model completely, as detailed in section 4

The sequence used to issue access to TLS symbols in the pre-existing general dynamic model[2] is as follows:

```
        ldr     r0, .Lt0
.L1:    add     r0, pc, r0
        bl      __tls_get_addr(PLT)
.Lt0:
        .word   variable(tlsgd) +
                       (. - .L1 - 8)
```

Our new model then states:

```
        ldr     r0, .Lt0
.L1:    bl      variable(tlscall)
.Lt0:
        .word   variable(tlsdesc) +
                       (. - .L1)
```

The `tlsdesc` relocation in `.Lt0` gives the pc-relative address of the TLS descriptor representing the thread-local variable we're interested in. The addend of the relocation is incomplete in order to properly allow linker relaxations, as stated in section 3.5 and must be adjusted at link time.

The `tlscall` relocation is resolved to the address of a linker-defined trampoline, detailed in sections 3.3 and 3.4.

In case of Local Dynamic access model, to avoid the definition of new relocations, the linker defines for all modules that have a TLS section a hidden per-module symbol called `_TLS_MODULE_BASE_` that denotes the beginning of its TLS section. We're then able to compute the address of this special symbol, which then serves as a base address to be added to the offset of any subsequently accessed symbol within this module in his address computation.

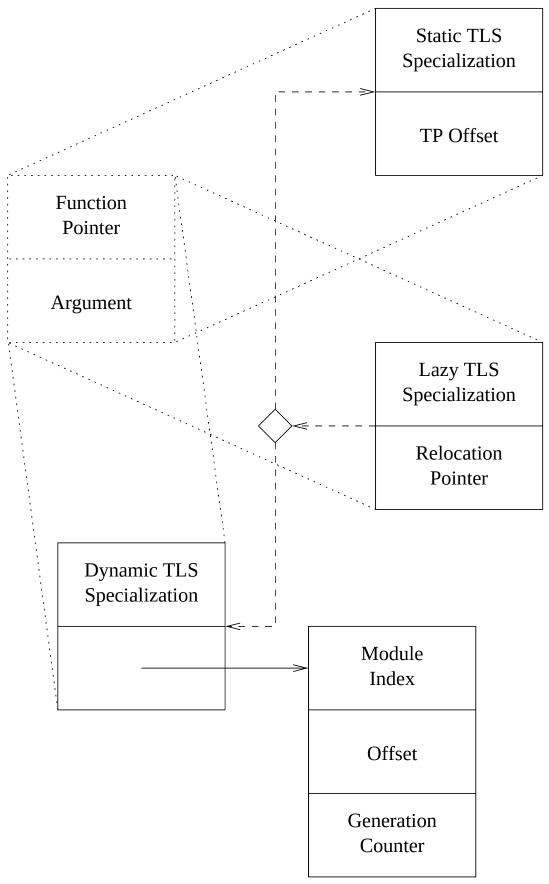

Figure 2: General structure of a TLS Descriptor, with 3 different specialization types, for Static and Dynamic TLS, and Lazy TLS that decays to one of the other two on the first use. Our model also has one more specialization that handles weak undefined symbols, not shown in the figure.

3.1 Selecting specializations at run time

When a module is loaded, whether this happens at startup or in a run-time call to `dlopen`, the dynamic loader is provided with enough information to devise a more efficient way to resolve a TLS variable address.

It has to select one of a set of possible specializations through which subsequent access to this variable may go. In our model, the possible specializations are:

Static Specialization being the most simple one, covering the case of initial modules. As this resolver's argument is the GOT address of the entry which contains the thread pointer offset of this symbol address, this specialization simply loads this value in register `r0` and returns

Dynamic Specialization happening now solely on *dlopened* modules, involving the call to `__tls_get_addr`. The specialization checks whether the module's generation count is current enough and the TLS block for the module is allocated, then calling `__tls_get_addr` if there is really need to. If it isn't, the thread pointer offset of the variable address is automatically returned.

Weak Undefined Symbols Current model differs from the expected behaviour of symbols falling in this class. This specialization is meant to provide TLS weak undefined symbols the same semantics they would get as a normal symbol, namely, getting a `NULL` pointer as the address of their variables. To achieve this, the specialization returns the negated value of the thread pointer, in order to nullify the symbol when added to it.

Lazy Specialization which allows an access to be resolved lazily instead of at startup. Lazy processing of relocations are detailed in section 3.6

3.2 Linker Relaxations

The design we devised for ARM TLS descriptors implementation may allow linker relaxations from the dynamic access models to the exec ones, removing overheads that in general cannot be avoided in the dynamic cases.

The result of relaxing our proposed dynamic sequence to Initial Exec is:

```
        ldr     r0, .Lt0
.L1:    ldr     r0, [pc, r0]
.Lt0:
        .word   variable(gottpoff) +
                       (. - .L1 - 8)
```

The branch and link instruction turns into a load. The addend of the relocation must be adjusted by the linker so as to provide the correct offset for it to be loaded relative to the instruction at `.L1`. Currently, our proposal is pretty much the same as that the current model states.[10]

And by relaxing to Local Exec, we get:

```
        ldr     r0, .Lt0
        nop
.Lt0:
        .word   variable(tpoff)
```

During a relaxation, the linker must not remove code, so as to preserve sizes, addresses

[2]The ARM ABI also defines a slightly modified version of `__tls_get_addr`, that expects the address to be relative to the link register at the time of the call, thus making the sequence one instruction smaller.

and offsets involved in operations that the compiler or the assembler may have already computed. Therefore, the branch and link instruction is turned into a `nop`. In all but this regard, the model is exactly what the current Local Exec model states.[10] As the addend of the `gottpoff` relocation is just a helper to the address computation, not an offset to be applied to the address of the variable, relaxing to `tpoff` discards it entirely.

3.2.1 Returning TP offsets

Current ARM TLS ABI states that access going through the General and Local Dynamic access models returns the final address of the variable to be accessed, while the Initial and Local Exec ones have to add the value held by the thread pointer in order to get it [10]. This design prevents linker relaxations to more efficient models when linking an executable. In our design, all accesses returns the variable's offset to the thread pointer, and have therefore to get the tp value added to it. This seldom imposes any overhead since ARM provides a register+register indirect addressing mode.

3.3 Trampoline

Due to size restrictions that may be present in embedded systems, we want to achieve efficient TLS access while minimizing the number of instructions emitted. By looking at the two desired relaxations described in section 3.2, one may conclude that no less than two instructions may be used by the dynamic cases, as doing so would prevent relaxation to the Initial Exec model.

Loading the address of the specialized function would usually incur in loading its pc-relative address from the constant pool, adding `pc` to it, loading the address of the chosen specialization from the GOT entry and jumping to it; too complex a task to be done by two instructions alone.

We thus define a per-module trampoline, that gives us the ability of accessing a variable with a two-instruction sequence. The complete sequence is indeed present and executed, but as it's put in a single location, we can thus achieve significantly size reduction, specially for a high number of accesses.

Given this, after the branch instruction that receives the `tlscall` relocation, the following piece of code gets hit:

```
__tls_trampoline:
        add      r0, lr, r0
        ldr      r1, [r0, #4]
        bx       r1
```

As detailed in section 3, the base address the trampoline gets after the add instruction is the GOT address of the TLS descriptor for this symbol, which is also the address of the parameter to the specialization. The load instruction then gets the specialization address from the next word and jumps to it.

3.4 Inlining the Trampoline

A trade-off between code size and performance is possible by inlining the trampoline described in section 3.3. By avoiding the branch instruction to the trampoline address and enabling better scheduling, we can expect the code to run faster

To provide the ability of inlining the trampoline, the compiler should be able to generate an instruction sequence that does the same job as the trampoline would have otherwise done. Such an instruction sequence may be:

```
        ldr     rt, .Lt1
.L1:    add     rx, pc, rt
        ldr     ry, [rx, #4]
        [mov    r0, rx]
        blx     ry
.Lt1:
        .word   variable(tlsdesc) +
                        (. - .L1)
```

In the previous and next examples, the `mov` instruction in brackets denotes that the specialization parameter may be a register other than `r0`, and must thus be copied to it if needed. Note that the addend is also incomplete here.

In order to keep the ability to relax the code sequence, the instructions must be annotated, as follows:

```
        ldr     rt, .Lt1
.tlsdescseq variable
.L1:    add     rx, pc, rt
.tlsdescseq variable
        ldr     ry, [rx, #4]
        [mov    r0, rx]
.tlsdescseq variable
        blx     ry
.Lt1:
       .word   variable(tlsdesc) +
                       (. - .L1)
```

Note that we do not force the use of specific registers, other than `r0` for the argument to the resolver (see resolver functions below), granting the compiler the ability to choose the best possible register allocation. There is no requirement that the instructions be issued in this particular sequence either, or that no other instructions be interspersed, or even that the values not be reused when it makes sense. It is even permitted for different registers to be used where the specification above implies a single register to be used, if the value is copied from one to the other. Such copies need not be annotated.

3.5 Addend Adjustments

Depending on whether the trampoline is inlined or not, we use different methods to compute the absolute address of a TLS descriptor.

The out-of-line trampoline adds `lr` to the relative address it is passed in `r0`, formerly loaded from the `tlsdesc` constant pool entry, where `lr` contains an address that is one instruction past the branch and link instruction annotated with the `tlscall` relocation, whereas the inline trampoline adds `pc` to the relative address loaded from the `tlsdesc` constant pool entry, where `pc` contains an address that is two instructions past the address of the instruction that refers to it.

Each case requires `tlscall` to be adjusted differently, even when the sequence happens to be relaxed, which makes matters more difficult as the offsets that have to be different before relaxation need to become the same after relaxation.

The solution that avoids the need for distinct relocation types for inline and out-of-line trampolines is to provide the linker with enough information for it to make the correct decision. We thus emit the relocation with an addend that provides the relative location of the instruction that is going to use the result of the relocation.

If it is a call instruction, presumed to be annotated with a `tlscall` relocation, the linker resolves the relocation such that its result, added to the `lr` value set by the call instruction, yields the address of the TLS descriptor, i.e., it subtracts 4 from the addend.

Otherwise, it computes the relocation result in such a way that adding its result to the `pc` value at the referenced instruction yields the address of the TLS descriptor, i.e., it subtracts 8 from the addend.

3.6 Lazy relocations

Although lazy relocation processing is very often applied to function calls, it is never applied to data accesses, since there is no transfer of control involved, and introducing it would render the access model too costly in terms of performance. However, in a TLS address resolution, a control transfer is indeed involved [4], which makes lazy processing of our newly-introduced relocations highly desirable, for getting more efficient program loading

The ability to relocate lazily is closely tied to the ability of reading and storing two got words atomically, as we would otherwise leave the GOT in an inconsistent state during the function real address resolving

Unlike FR-V [12], ARM processors provides no real atomic double-word memory operation [3], so lazy relocation processing has to go through the acquisition of a dynamic loader lock.

When the lazy resolver function is called, it starts by checking if there is still need to resolve the symbol value, by comparing the value present in the TLS descriptor with it's caller's address. If it finds such a need, it holds the lock, and fill the TLS descriptor entry with a temporary resolver address; a placeholder for the now waiting threads that came late to resolution processing.

It then proceeds with resolving the symbol the relocation refers to, deciding which of the available specializations is to be used and setting up the TLS descriptor according to the decision, such that subsequent calls involving the same TLS descriptors go straight to the most efficient specialization. Finally the lock is released and all the other threads possibly waiting on it awaken, being the resolver last step to jump to the specialization code chosen. This simple mechanism is indeed the same used by the i386 and x86_64 processors [11]

[3]The ldrd/strd instruction pair are atomic regarding instruction ordering, but gives no behavioral guarantees in the case an interrupt occurs, for example.

The lazy resolver needs to obtain the relocation index and the `_GLOBAL_OFFSET_TABLE_` address in order to perform lazy relocations. The relocation index can easily fit in the argument portion of the descriptor, but loading the GOT address in a register prior to calling the TLS resolver was deemed as too much overhead, since it is only necessary for the first time a descriptor is used.

We have therefore introduced another per-module trampoline, that TLS descriptors eligible for lazy relocation get as their resolver. The address of this trampoline is communicated to the dynamic loader by means of a dynamic table entry: `DT_TLSDESC_PLT`. It loads the address of the actual resolver from a GOT entry, whose address is informed to the dynamic loader with another dynamic table entry: `DT_TLSDESC_GOT`. The dynamic loader is responsible for filling in the named GOT entry with the address of the actual TLS lazy resolver address, whose name can thus remain internal to the dynamic loader.

4 Implementation issues

Two new options were added to GCC for controlling TLS descriptors behavior, namely:

- `-mtls-dialect`, which drives the selection of an access model for this object, receiving the options `arm` and `gnu`, the old and new models respectively.

- `-mtls-inline-trampoline`, a binary option that enables the use of the inline trampoline performance optimization

4.1 ABI addenda

We have added the following new relocation types:

R_ARM_TLS_GOTDESC is emitted by the assembler as `(tlsdesc)`, being resolved by the linker to the address of the first GOT word in the TLS descriptor. Upon facing this relocation, the linker must first adjust the relocation addend accordingly, as described in section 3.5

R_ARM_TLS_CALL is emitted by the assembler as `(tlscall)`, and is resolved by the linker to the address of the trampoline.

R_ARM_TLS_DESCSEQ is emitted by the assembler, annotating the instructions in the call path for the inline trampoline. Its rationale and usage are described in section 3.4.

R_ARM_TLS_DESC is emitted by the linker in response to R_ARM_TLS-_GOTDESC and/or R_ARM_TLS_CALL and gets no addend. The dynamic loader then resolves it to the specialization function it decided to use for access to this symbol.

As the current ARM ABI states preference for the use of REL relocations [13], so are our newly defined relocations.

Furthermore, our implementation also makes uses of two newly defined [11] Dynamic Table entries, `DT_TLSDESC_PLT` and `DT_TLSDESC_GOT`, which have their usage and motivation are explained in section 3.6, and a new hidden symbol, namely `_TLS_MODULE_BASE_`.

4.2 Specialized calling conventions

Besides specifications of where arguments are passed and where return values are stored, another important aspect of calling conventions is that of defining which registers a function can modify without preserving (caller-saved or call-clobbered), and which have to be saved before they can be modified (callee-saved or call-preserved).

Since in this work we are defining a new interface for `__tls_get_addr` specializations, we might as well define the conventions regarding preserved registers to privilege the most common cases. We have thus defined that the specializations are to preserve all registers they modify, including the usually call-clobbered ones. Three registers are exception to this rule, namely `r0`, expected to return the value we need; `r1`, expected (but not required) to hold the throw-away resolver address; and the processor flags, that would be too expensive to save and restore, compared with the benefit it might bring.

5 Performance

To figure out the speedup obtained by our model, we ran an adaptation of the benchmark that the x86 processors went through [4]. Tests have been conducted on an OMAP1611 ARM board, featuring an ARM 926 processor, running a 2.6.12 series kernel. A major difference from previous tests is the method we used access times. Unlike x86 [14] and ARM Xscale [15] processors, ARM 926 lacks a benchmark cycle measuring instruction. Time measurements were then made using libc's `clock()` call, giving us an approximation of the time used, in microseconds resolution. We thus run an access pattern repeatedly within a loop, and then divide the total time by the number of iterations to estimate the per-iteration time, expecting the error incurred by the coarse measure to be negligible compared to the total run time.

Another important difference is the absence of the distinction between internal and external timing, as the system call overhead would dominate the access time in an internal timing scenario.

A comparison is shown between a shared library compiled with the `arm` TLS dialect, and our newly proposed `gnu` one. Execution series are divided in five categories, as follows:

- **SE tests** access a variable with the Initial Exec model.
- **SR tests** access a variable in the General Dynamic method, which is aliased to a variable previously accessed with Initial Exec. Wherever applicable, we expect access to this variable to be relaxed by the linker.
- **SG tests** access a variable defined in the main executable, thus living in the Static TLS block. There is no way to be aware of it at link time, so the General Dynamic model is expected to be chosen. Our model, however, should be able to notice it at load time, and fill the GOT entries with the static specialization.
- **DG tests** access a variable defined in the shared library. As the libraries are `dlopened` during the execution, the call to `__tls_get_addr` has to be issued. In our model, it corresponds to using the dynamic specialization.
- **DC tests** access a mixture of the previous variables. It aims to simulate a more complex environment.

All these tests are run in four different configurations, varying two dimensions. The first one is the access operation, either a full load, in which we use the variable contents, or just an address resolution. Besides that, we simulate situations in which we put pressure in the register allocator by marking all available registers as allocated, or no pressure at all.

Table 1 shows the tests results obtained in those scenarios. **Ov** and **Nv** are the results with the old and new model respectively, with all registers marked as used. Accordingly, **On** and **Nn** shows the results with no pressure over the allocator. Results are shown for both accessing methods, labeled **load** and **addr**

Figures 3, 4, 5, 6 and 7 shows the average time comparison between old, on the left, and new model, on the right, for each of the five scenarios tried. In labels, *N* stands for no pressure over the register allocator while *V* for its opposite. *L* stands for a variable-load test, while *A* an address-computing only. The dashed line shows the estimated overhead, computed from a run with the same number of iterations of a configuration that does not call an actual access function.

5.1 Analysis

From the data in table 1 and the five charts presented, one can see that our model consistently outperforms current one. SE tests can be thought as a lower bound as they access a variable defined in the module itself, and as expected, went to very similar results in both

Model	Op	On	Nn	Ov	Nv
SE	load	505	505	578	578
	addr	449	443	609	625
SR	load	1301	521	1377	588
	addr	1273	494	1424	625
SG	load	1298	589	1366	662
	addr	1274	565	1424	703
DG	load	1304	863	1377	925
	addr	1279	838	1429	975
DC	load	2243	1036	2300	1055
	addr	2097	970	2304	1202

Table 1: Experimental results showing a series of 100000000 runs. The estimated overhead was found to be 120 ns. Values presented are the average in the set, in nanosecond scale.

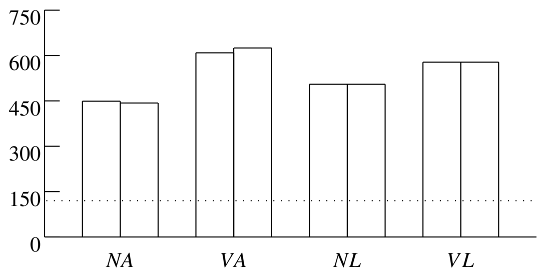

Figure 3: Comparison chart of access times in nanoseconds between the old and new model for the SE tests

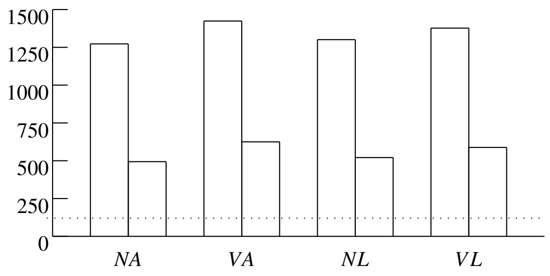

Figure 4: Comparison chart of access times in nanoseconds between the old and new model for the SR tests

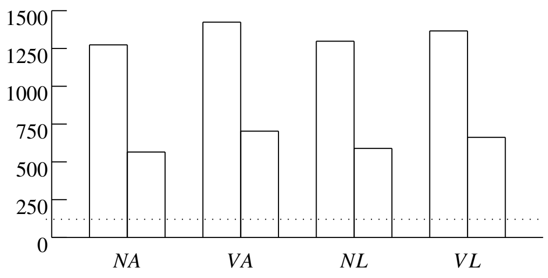

Figure 5: Comparison chart of access times in nanoseconds between the old and new model for the SG tests

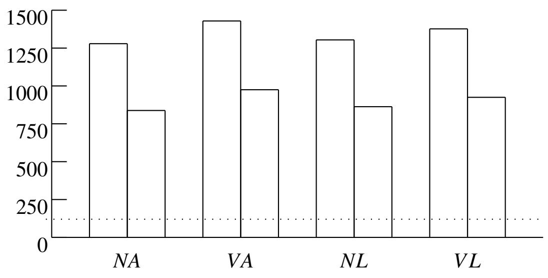

Figure 6: Comparison chart of access times in nanoseconds between the old and new model for the DG tests

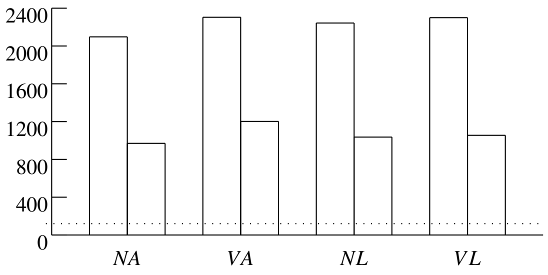

Figure 7: Comparison chart of access times in nanoseconds between the old and new model for the DC tests

models. SR tests delivers a speedup up to 2.5 times, due to the now-present ability to relax a code sequence to a more efficient access model. For those we consider to be the common case, namely SG tests, we deliver results at least 2.2 times better than the old model did.

Experiments also show that our model's dynamic specialization leverages better results than calling `__tls_get_addr` directly. While at a first glance it may seem illogical, we can avoid the call to `__tls_get_addr` in the most common situation, which is the one simulated here, where runtime load or unload of a module is rare resulting in the DTV being sufficiently recent [4]

6 Future Work

The implementation on the GNU toolchain is nearly finished and about to be submitted upstream. Furthermore, support to more widely used C libraries in the embedded world, such as the uClibc would certainly be more than welcome, and may get into the spotlight as its NPTL support matures.

As the TLS descriptor access model proves itself successful, other architectures may have enough stimuli for embracing its benefits. In this implementation, though not the main focus, we tried to keep the design as friendly as possible for a future Thumb ABI extension.

More experiments also have to be conducted to verify if our predictions regarding the speedups by inlining the trampoline are true, and by how much.

7 Conclusion

Although slightly different due to architecture specificity, the TLS descriptors implementation could rely on, and in some points extend, most features found in previous implementations.

We were able to add more flexibility to the ABI for the symbols using our new model. Relaxation to more efficient access models were not possible using the original TLS access model, being now completely feasible both to Local or Initial Exec. Generating code with the expected behavior for Weak Undefined Symbols is now also possible, with minimum effort/overhead.

Such flexibility increase came along with a substantial performance improvement, which is enough to consider the model worthwhile. With this in mind, we expect TLS descriptors to be the de facto choice for TLS symbols handling.

Acknowledgements

Glauber de Oliveira Costa thanks his family for the endless support, Professor Guido Araújo for the opportunity of using time of his course for this implementation; INdT for providing fancy ARM boards; The guy (or entity) who invented caffeine, whoever he is, for keeping us all awake during hard times; Chuck Norris, for having counted to infinity (twice). [16]

Alexandre Oliva thanks Red Hat, for having permitted him to pursue the initial ideas of TLS Descriptors. He also seconds Glauber's gratitude towards INdT and professor Guido. Not to feel left behind, he also thanks the entity that came up with coffee and guaraná. Drink moderately!

References

[1] Herb Sutter. The free lunch is over: a fundamental turn toward concurrency in

software. *Dr. Dobb's Journal*, 30(3), 2005. http://www.gotw.ca/publications/concurrency-ddj.htm.

[2] Kunle Olukotun and Lance Hammond. The future of microprocessors. *ACM Queue*, 3(7):26–34, September 2005.

[3] Portable Applications Standards Committee of the IEEE Computer Society and The Open Group. Portable Operating System Interface (POSIX), The Base Specifications. IEEE Std 1003.1, 2004. Issue 6, Incorporating Technical Corrigendum 1 and Technical Corrigendum 2.

[4] Ulrich Drepper. ELF Handling for Thread-Local Storage. http://people.redhat.com/drepper/tls.pdf, February 2003. Version 0.20.

[5] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann, October 1999.

[6] Hans-J. Boehm. Fast multiprocessor memory allocation and garbage collection. Technical Report 165, HP Labs, 2000.

[7] David A. Patterson and John L. Hennessy. *Computer Organization and Design*. Morgan Kaufmann, third edition, 2005.

[8] GCC 3.3 Release Series Changes, New Features, and Fixes. http://gcc.gnu.org/gcc-3.3/changes.html.

[9] Alexandre Oliva and Guido Araújo. Speeding up thread-local storage access in dynamic libraries. In *Proceedings of the GCC Developer's Summit*, pages 159–178, June 2006.

[10] Lee Smith. Addenda to, and Errata in, the ABI for the ARM architecture. http://http://www.arm.com/miscPDFs/8693.pdf, May 2006.

[11] Alexandre Oliva. Thread-Local Storage Descriptors for IA32 and AMD64/EM64T. http://people.redhat.com/aoliva/writeups/TLS/RFC-TLSDESC-x86.txt, October 2005. Version 0.9.4.

[12] Alexandre Oliva and Aldy Hernandez. The FR-V thread-local storage ABI. http://people.redhat.com/aoliva/writeups/FR-V/FDPIC-TLS-ABI.txt, December 2004. Version 0.22.

[13] Richard Earnshaw. ELF for the ARM architecture. http://www.arm.com/miscPDFs/8030.pdf, May 2006.

[14] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, volume 2: Instuction Set Reference. Intel Corporation, 2003.

[15] Intel Xscale Core - Developer's Manual. http://download.intel.com/design/intelxscale/27347302.pdf, 2004.

[16] Chuck Norris Facts. http://www.chucknorrisfacts.com.