Reprinted from the

Proceedings of the
GCC Developers' Summit

June 28th–30th, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon Incorporated*
C. Craig Ross, *Linux Symposium*

Review Committee

Ben Elliston, *IBM*
Janis Johnson, *IBM*
Mark Mitchell, *CodeSourcery*
Toshi Morita
Diego Novillo, *Red Hat*
Gerald Pfeifer, *Novell*
Ian Lance Taylor, *Google*
C. Craig Ross, *Linux Symposium*
Andrew J. Hutton, *Steamballoon Incorporated*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Speeding Up Thread-Local Storage Access in Dynamic Libraries

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

Guido Araújo
IC-Unicamp
guido@ic.unicamp.br

Abstract

As multi-core processors become the norm 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. This paper proposes an access model that enables dynamic libraries to take advantage of this fact, without giving up the ability to be loaded during program execution. This new model was implemented and tested on GNU/Linux systems, initially on the Fujitsu FR-V architecture, and later on IA32 and AMD64/EM64T, such that performance could be compared with that of the existing models.

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.

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.

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 reengineering 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 variables and even less efficient than access to 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, for reasons detailed in Section 2; for main executables, access can sometimes be just as efficient as accessing automatic or global variables. The mechanisms introduced in Section 3, based on the novel concept of TLS Descriptors[7, 8], yield a major speedup, that brings the performance of TLS access in dynamic libraries close to that of executables, as shown in Section 4. Section 5 summarizes the results with some final remarks and future directions.

2 Background

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`.

Every loadable module may define a memory address range delimiting its TLS segment. This range, after relocation processing, contains the memory image to be used to initialize the TLS block associated with that module, for each different thread.

For every thread, two data structures are allocated: a Static TLS Block and a Dynamic Thread Vector (DTV), as depicted in Figure 1.

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

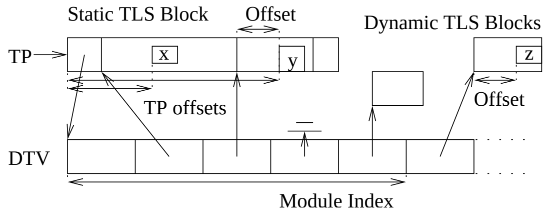

Figure 1: Per-thread data structures used to support TLS.

A reserved register, called the Thread Pointer (TP, for short), points to a base address within that thread's Static TLS Block. At a fixed relative location within the Static TLS Block lies a pointer to the DTV. The DTV, in turn, starts with a generation counter, followed by pointers to TLS Blocks. For every module containing a TLS segment, a module index is assigned, that indicates the entry in each thread's DTV reserved to hold a pointer to the TLS Block corresponding to that module.

The dynamic loader can use information about the TLS segments of all initial modules to lay out the Static TLS Block. Each thread's static block will contain TLS blocks for all initial modules. Using the same layout for all threads implies that the relative locations, in the Static TLS Block, of the initial modules's TLS blocks's are the same across all threads, thus enabling not only efficient code generation for some TLS access models, but also the optimization proposed in Section 3.

2.1 Access Models

If a main executable contains a TLS segment, the dynamic loader not only reserves the first entry in the DTV for it, but also lays out the Static TLS Block in such a way that the offset from the TP to the executable's TLS block is a constant computable at link time. The exact location of the executable's TLS block within the Static TLS Block only depends on the size and alignment requirements of the executable's TLS segment, and conventions set by the Application Binary Interface (ABI) of the hardware architecture and operating system. Since the linker can compute the offset from the TP to the executable's TLS block, and the relative location of a variable defined within this block, it can compute the exact TP offset of such a variable (say, variable x in Figure 1), and use that as a displacement from the TP to access the variable. This access model is known as Local Exec. It is the most efficient, but least general, access model, since only the main executable can use it. In theory, all initial modules could use it, but this would require text segments to be modified at dynamic relocation processing time, and modifying text segments is highly undesirable, mainly because it prevents page sharing across multiple processes, which is what shared libraries are supposed to enable.

An example of computing the address of a variable `var` into register `reg` using the Local Exec access model, in low-level pseudo code, is given below. `TPoff` is a functional notation to denote the TP offset of a variable.

```
let reg ← TP + TPoff(var)
```

Accessing thread-local variables that are not defined in the main executable preclude the use of the Local Exec access model. The main executable, however, can still take advantage of the fact that every dynamic library it depends on, that might provide the variable it wants to access, is an initial library, and therefore its relative location within the Static TLS Block is a run-time constant, which holds for variables x and y in Figure 1. Emitting a relocation to get the dynamic loader to compute this run-time constant and store it into a Global Offset Table (GOT) entry, and using this constant, loaded from the GOT, as an offset from the TP to access the variable, is called the Initial Exec access model. Under certain circumstances, it may be used in dynamic libraries as well, but it may come at the cost of being unable to dlopen such libraries. The use of indirection through the GOT, allocated in the data segment, not only retains the ability to share pages of code, but also merges all the dynamic address computation related with a symbol into a single location, reducing the number of dynamic relocations needed.

An example of computing the address of variable `var` into register `reg` using the Initial Exec access model follows. `GOT`, in such low-level pseudo code, denotes a reserved register or some PC-relative addressing mode that yields the GOT base address. `GOTTPoff` denotes the offset of a GOT entry that, at run time, will hold the TP offset of a variable.

```
load reg, GOT[GOTTPoff(var)]
let reg ← TP + reg
```

The other two access models, General Dynamic and Local Dynamic, require the (implicit) use of the DTV. Both access models involve calling a function, normally called `__tls_get_addr`, to obtain a thread-local address. Function `__tls_get_addr` requires two pieces of information to compute the requested address: a Module Index and an Offset within the module's TLS segment, as depicted in Figure 1 for variable `z`. These two pieces of information are normally computed by the dynamic loader, in response to relocation entries that request them to be stored in the GOT. An example of the use of the General Dynamic access model is given below, using adjacent GOT entries and passing it by reference in a register. Other implementations use independent GOT entries for the two values, and/or pass them by value. `GOTModIdx\&Off` is a functional notation to denote the offset of a GOT entry that, at run time, will hold a Module Index followed by a corresponding Offset.

```
let reg ← GOT + GOTModIdx&Off(var)
call __tls_get_addr
```

Local Dynamic is a variant of General Dynamic that calls the function to compute a base address, normally by passing the function a zero offset. Having obtained the base address of a module's TLS block with a single call, the Local Dynamic access model then uses variables's offsets to access them using the same base address. The offsets can all be computed by the linker, since they are a local property of the module. An example follows, in which `GOTModIdx` denotes the GOT offset for an entry that, at run time, will hold the Module Index and a zero offset, and `ModOff` represents the Offset of a given variable.

```
let reg ← GOT + GOTModIdx()
call __tls_get_addr
let reg1 ← reg + ModOff(var1)
let reg2 ← reg + ModOff(var2)
```

2.2 Dynamic behavior

At thread creation time, the DTV is initialized such that every entry corresponding to an initial module points to a TLS block within the Static TLS Block, like the second and third slots in the DTV in Figure 1, and all other entries are marked as not allocated, like the fourth slot. Entries for dlopened modules have to be assigned on demand to TLS blocks allocated dynamically, as depicted by the two Dynamic TLS Blocks in the figure. Dynamic allocation is necessary because multiple threads may already be running at the time a new module is loaded into a process. Function `__tls_get_addr` is responsible for the run-time maintenance of the DTV.

The generation counter in the DTV is used to keep track of such dynamically-allocated TLS blocks: every time a dlopened module with a TLS segment is loaded or unloaded, a global generation counter is incremented. Function __tls_get_addr checks whether the DTV generation counter is up to date every time it is called. If the DTV is found to be out of date, the function may have to release the memory associated with its outdated entries, to dynamically resize it, and to set any released or newly-created entries to the unallocated state.

Once the DTV is up to date, if function __tls_get_addr finds that the requested DTV entry is not allocated, it allocates the necessary storage, initializes it with the contents of the TLS segment from the corresponding module and sets the DTV entry to the allocated address. At last, it loads the module's TLS block's base address from the corresponding DTV entry and adds to it the variable offset it was passed as argument, returning the result.

3 Optimization

Let us first investigate why __tls_get_addr is perceived as so slow, and then proceed to introducing the optimization subject of this paper.

3.1 Inefficiencies in __tls_get_addr

It might seem that the dynamic access models should not be so expensive, since in the most common case, the run-time behavior of function __tls_get_addr will involve two test-and-branch sequences, with branches predicted not taken, followed by offsetting the base address already loaded for the second test by the amount given as an argument, as in the low-level pseudo code below. DTVoff denotes the offset from the TP to the DTV address stored in the Static TLS block; DTVGCoff, the relative location of the generation counter in the DTV, normally 0; DTVentrysize, the size of a DTV entry; arg1 and arg2, the module index and the offset, respectively; result, the register in which __tls_get_addr returns its result.

```
load reg1 ← TP[DTVoff]
load reg2 ← generation_counter
branch to slow path 1 if reg1[DTVGCoff] < reg2
load reg2 ← reg1[arg1 × DTVentrysize]
branch to slow path 2 if reg2 == UNALLOCATED
let result ← reg2 + arg2
return
```

The first test, however, involves a global variable, the global generation counter. Accessing a global variable can be relatively expensive in such a simple function, since it may require setting up the GOT register to compute its address, if PC-relative addressing is not available.

A bigger performance penalty follows from the compiler's inability to shrink-wrap functions[9, 10], namely, to avoid saving and restoring registers, and even setting up a stack frame, in the fast path that issues no function calls and needs only two scratch registers. Since the slow paths issue function calls, compilers will generally set up a stack frame for the entire function, and since such paths are complex, possibly requiring multiple registers, several such registers have to be saved and restored every time the function is called, even though they are seldom actually used.

Although some register saving and restoring performance can be recovered by means of shrink-wrapping, compilers cannot help the fact that the definition of __tls_get_addr, in the dynamic loader, is publicly visible and not actually known before run time, so the compiler must assume it complies with the platform-defined calling conventions. ABI-defined custom calling conventions for this function could shift into the `__tls_get_addr` slow path the penalties involved with preserving registers that would otherwise have to take place in its callers.

Yet another performance penalty is related with the fact that `__tls_get_addr` is always called through Procedure Linkage Table (PLT) entries. Since it is defined in the dynamic loader, calls to it in other modules have to go through such an entry that loads the actual function address from the GOT and then jumps to it.

Without such inefficiencies, the instruction sequence above would be observed at run time. However, with all the inefficiencies, the dynamic instruction trace after an instructions that calls `__tls_get_addr` is as follows. Additional instructions, not present above, are *emphasized*. `GOToff(sym)` denotes the offset from the GOT to the address of symbol `sym`.

```
jump to address loaded from PLT GOT entry
set up stack frame
save call-preserved registers used in slow path
save and set up GOT register if needed
load reg1 ← TP[DTVoff]
load reg2 ← GOT[GOToff(generation_counter)]
branch to slow path 1 if reg1[DTVGCoff] < reg2
load reg2 ← reg1[arg1 × DTVentrysize]
branch to slow path 2 if reg2 == UNALLOCATED
let result ← reg2 + arg2
restore registers
destroy stack frame
return
```

Even if the compiler could be improved so as to avoid setting up a stack frame, the GOT-relative addressing mode to access the generation counter is unavoidable. As for the PLT entry, the additional jump could be avoided by using a call sequence in `__tls_get_addr` callers that referenced its GOT entry directly, precluding lazy relocation of this reference and, most often, requiring larger code size at all call sites, negatively impacting the instruction cache efficiency.

3.2 TLS Descriptors

From the previous paragraph, it would seem that improving the performance of the dynamic access models would not involve a change in the access models themselves, but rather in the compiler used to compile `__tls_get_addr`.

It is possible, however, to make them more efficient, by introducing specialized versions thereof for different situations, and by providing such specialized versions with additional information. Let us put aside for a moment the issue of how to get the most appropriate specialized version selected efficiently, and concentrate on the potential benefits first.

3.2.1 Improving Static TLS

One major shortcoming of `__tls_get_addr` is that it fails to take advantage of the fact that, to access the TLS block for an initial module, no tests are necessary. Since initial modules' TLS blocks are laid out as part of Static TLS Blocks, all threads' DTVs already contain the correct addresses in the entries corresponding to such modules, so it would suffice to dereference the DTV and add the variable offset.

However, it is possible to do even better in the Static TLS case: since the initial module's TLS block is at an offset from the TP that is the same for all threads, we can use the provision above of passing additional information to the specialized function and pass it this constant TP offset, instead of the then-unused module index. Thus, all this specialized function has to do is to add the module's TP offset to the TP, and then to the variable offset.

In a further step, this specialized function could take as arguments, instead of the TP offset and the variable offset, the precomputed result of adding them together. This specialized function is thus reduced to the following pseudo code:

```
let result $\leftarrow$ TP + arg
return
```

Selecting this specialized function reduces significantly the computation performed in the function, rendering its performance very similar to that of the Initial Exec or even Local Exec models, discounting the function call overhead. The use of this specialized version is the most significant improvement we have introduced, but there are additional minor improvements to follow.

One important point to consider is that all specializations must present the same interface, such that callers are totally unaware of which specialization is selected; such selection takes place at run time, at which point it is undesirable to modify code. Therefore, when we modify the interface of a specialization so as to take a single argument, we are either determining that none of the specializations can take more than one argument, or that this one specialization will ignore any additional arguments other specializations might require.

3.2.2 Returning TP offsets

On some architectures, register-plus-register indirect addressing modes is little or no more expensive than indirect addressing modes. On Fujitsu FR-V, for example, there is no single-register indirect addressing mode: loads and stores compute the address by adding a register to either another register or a constant displacement. On IA32 and AMD64/EM64T, on GNU/Linux, segment registers are used as TP, so an instruction with a single-register indirect addressing mode can be modified to use this register as an offset from the segment base address by using a 1-byte prefix, with no significant performance penalty.

On such architectures, it makes sense to arrange for the function to return not the address of the variable, but rather its TP offset. If it is also possible to arrange for the argument to be passed in the register used to hold return values, then the specialization optimized for Static TLS becomes a single return statement, as on FR-V. On IA32 and AMD64/EM64T, it could be possible to achieve the same, but at the expense of additional code at every call site to load the argument from memory. Thus, it is more efficient, in terms of code size, to leave it up to the specialized function to load it before returning.

3.2.3 Linker relaxations

TLS-related relaxations are always defined so as to turn accesses using dynamic access models into Initial Exec or Local Exec, when linking an executable. In general, the `__tls_get_addr` call sequence, including the instructions that set up the arguments, has to contain padding such that, if the linker relaxes the code to a more efficient access model, there is room for the instruction that adds the TP and the TP offset, regardless of whether it is the Local Exec link-time constant or the Initial Exec run-time constant loaded from the GOT.

The convention of returning the TP offset instead of the actual address simplifies linker relaxations, because the addition of the TP does not have to fit in the replacement sequence: it is already there, after the call sequence. So it suffices to arrange for the value loaded from the GOT, or the fixed constant used in Local Exec, to make it to the register in which the call would have returned the TP offset. With the reduced padding, code size is reduced, improving the efficiency of the instruction cache.

3.2.4 Avoiding unnecessary DTV updates

The use of a global variable, namely the generation counter, when testing whether a DTV is up to date, is not only a bad idea because of the potential performance hit associated with saving, setting up and restoring the GOT register.

The fact that some thread *A* may choose to dlopen or dlclose a module *a* may slow down another thread *B* that accesses TLS variables from module *b*. This occurs because the test in `__tls_get_addr` checks whether the DTV is up to date, and not whether it is recent enough to access a variable in the requested module.

While indexing some TLS module table to determine the generation count associated with a module could be feasible, it would significantly slow down the fast path. However, with our provision of passing additional information to the specialized functions, we can arrange to have the minimum generation count needed to access a module's TLS passed to a specialized function used to handle Dynamic TLS.

Since we have arranged for the Static TLS specialization to use a single argument, we can do the same for the Dynamic TLS specialization at hand. Since there is no way to avoid the requirement for the module index and the offset, however, in order to fit all this information in a single argument, the only solution is to use indirection.

Since Dynamic TLS is designed to be the rare case, allocating additional storage for references to such variables is not deemed unacceptable, so what we do here is to arrange for the Dynamic TLS specialization to be passed, as its argument, a pointer to a data structure containing not only the module index and the offset, but also the generation counter needed by the module. The specialized function can thus avoid the need for the GOT register in the fast path, using for the test the generation counter stored in this data structure passed as its argument, also avoiding DTV updates that would not affect its ability to access the requested module.

On Fujitsu FR-V, a particular detail of the ABI[11] required an additional field in this data structure. The ABI requires the GOT register to be set up for a function not by the function itself, but rather by its caller. Since no specializations of TLS calls would require the GOT register in their fast paths, we have arranged for the argument data structure to contain the GOT pointer the specialization may need.

An additional micro-optimization, applied on FR-V, is to arrange for this data structure to contain not the module index, but rather the offset into the DTV where its entry is stored. This saves a shift-left instruction in the fast path of the specialized function, because FR-V does not have an addressing mode that adds an index register multiplied by a constant to a base register.

3.2.5 Specialized calling conventions

The IA32 version of `__tls_get_addr` on GNU/Linux has traditionally used custom calling conventions in that its arguments are not passed on the stack, as usual, but rather on registers. This should also be the case of specializations of this function.

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).

The most common TLS cases in code compiled for dynamic libraries, namely Static TLS specialization and relaxation for main executable, can assume that, in a TLS call instruction or its replacement, no register is modified other than the one holding the resulting address or TP offset.

Only the Dynamic TLS specialization needs a pair of temporary registers for the fast path, and potentially several other registers for the slow path.

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 other than the return value, such that TLS calls can be modeled like simple loads, enabling the full register set to be used without concerns about preserving registers across such calls. This requires that, when the slow path of the Dynamic TLS specialization issues calls to other functions, it preserves all registers that they might modify. Since it is the slow path, and it has so much work to do anyway, this additional work is insignificant. Unfortunately, this decision also affects the fast path, in that it has to preserve the two scratch registers it needs, but since Dynamic TLS is assumed to be the uncommon case, privileging the Static TLS case is a reasonable decision.

3.2.6 Selecting specializations at run time

Now that we have established that both specializations work with a single argument, and defined that they should use customized calling conventions to do their jobs, we are ready to specify how the appropriate specialization is to be selected and called.

In the existing dynamic access models, two GOT entries are needed to hold the arguments to `__tls_get_addr`. Since for the specialized versions we can use only one, we can use the other to hold the address of the specialized function. Then, we arrange for the code, that used to call `__tls_get_addr`, to call the function whose address is stored in that location.

As a general rule, we can store the function address at the GOT entry that would, in the traditional access model, contain the module index, and the argument to the function, in the GOT entry that would contain the variable offset. Since, for a given module, the decision on whether its TLS block can be accessed with the Static or the Dynamic specialization is the same for all variables in the block, this general rule works even for ABIs that enable the module index and the variable offset to be in non-adjacent entries, with potential use of the module index entry to access multiple variables.

The machines on which the new access model was implemented, however, all use adjacent GOT entries, since they make the code much simpler, at the expense of additional GOT space due to the multiple copies of the the same module index. Nevertheless, the absence of such sharing enables lazy processing of relocations, as detailed in Section 3.2.8. When the entries are adjacent, they form a data structure that we call TLS Descriptor, named after Function Descriptors, present in ABIs such as IA64's[12], PPC64's[13] and FR-V's[11], that contain a

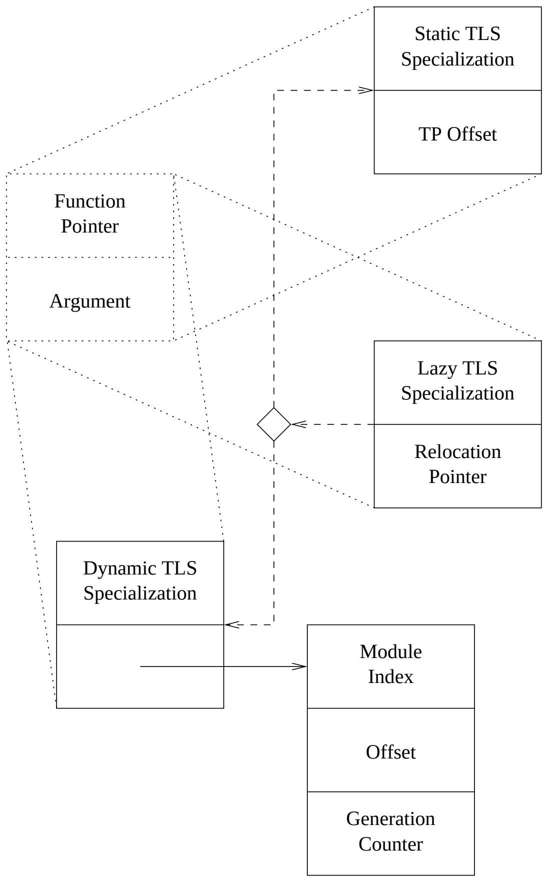

Figure 2: General structure of a TLS Descriptor, with 3 different specialization types, for Static and Dynamic TLS, and Lazy TLS that decays to one of the other two on the first use.

function's entry point and a context pointer, e.g., the GOT pointer to be used by the function. TLS descriptors also take two words, but, instead of a context pointer, their second word contains an argument to the function whose pointer is in the first word, as depicted in Figure 2.

Fujitsu FR-V has never had a traditional TLS ABI, since it was already designed taking advantage of the new access model, but we can imagine that, if it had, the instruction sequence would be as follows.

```
sethi.p #gottlsgdhi(var), gr8
setlo #gottlsgdlo(var), gr8
ldd #tlsgd(var)@(gr15, gr8), gr8
call __tls_get_addr
```

The `ldd` instruction loads into the pair of registers starting at `gr8` the pair of words starting at the address obtained by adding `gr15`, the GOT pointer, and `gr8`, whose value was set to the linker-computed displacement for the GOT entry containing the module index and the variable offset. In the actual FR-V TLS ABI, the call sequence is as follows.

```
sethi.p #gottlsdeschi(var), gr8
setlo #gottlsdesclo(var), gr8
ldd #tlsdesc(var)@(gr15, gr8), gr8
calll #gettlsoff(var)@(gr8, gr0)
```

The variation here is mainly from relocations that reference a TLS Global Dynamic GOT entry to those that reference a TLS Descriptor GOT entry, and the last instruction, that is a call to a named function in the former, that goes through a PLT entry, and a call to a given address in the latter, that goes straight to the specialization. The address was loaded into `gr8`; `gr0` is fixed at zero.

On GNU/Linux IA32, the difference is a bit more significant. The current TLS ABI specifies the following sequence for the General Dynamic access model.

```
leal var@TLSGD(,%ebx,1), %eax
call ___tls_get_addr@PLT
```

This uses an extraneous addressing mode for `leal`, equivalent to `(%ebx)`, but longer, making the instruction long enough for the relaxation replacement, that takes 12 bytes. Our version, however, is as short as 8 bytes for the call sequence, although it requires an additional byte for the segment prefix to the load or store instruction that uses the resulting offset.

```
leal var@TLSDESC(%ebx), %eax
call *var@TLSCALL(%eax)
```

Note that `var@TLSCALL` is just an annotation to aid linker relaxations, such that the two instructions can be scheduled apart. The actual instruction encoding is the two-byte indirect call, that calls the function at the address stored at the memory location whose address was computed into `%eax` by the `leal` instruction. The called specialization knows that, at its entry point, `%eax` points to the TLS descriptor, so it can load its argument from the descriptor.

On AMD64/EM64T, the original call sequence contains several meaningless padding prefixes to make room for relaxation substitutions, as follows.

```
.byte 0x66
leaq var@TLSGD(%rip), %rdi
.word 0x6666
rex64
call __tls_get_addr@PLT
```

Our improved call sequence follows the very same pattern as IA32, with the difference that GOT accesses do not involve a fixed register, but are PC-relative, and register and addresses are 64-bits wide. While the above takes 16 bytes, the following takes as little as 9, plus one for the byte prefix in actual accesses.

```
leaq var@TLSDESC(%rip), %rax
call *var@TLSCALL(%rax)
```

3.2.7 DTV compression

When this new access model is used, and the traditional one is not (i.e., `__tls_get_addr` is never called directly), it is possible to remove all static entries from the DTV, since they are never used. Since we know that every access to Static TLS will go through the static specialization, that does not use the DTV, entries for such modules can be entirely removed, enabling the initial DTV to be trivially set up.

This offers a slight speed up in thread creation for processes that have multiple initial modules with TLS segments, potentially saves memory by delaying the need for dynamically growing the DTV, and enables the DTV to be reduced by half, since its current definition reserves a word in every entry to tell whether it is static or dynamic when the time comes to release that entry and free up its storage.

Even when the traditional dynamic TLS access model is used, it is possible to enable this DTV compression, as long as the index range reserved for initial modules can be easily distinguished from that of dlopened modules, for example, by having the most significant bit set. `__tls_get_addr` would then have to recognize this case and use an alternate code path that, instead of relying on the DTV, obtained the module's constant TP offset from a separate table.

3.2.8 Lazy relocations

Processing relocation entries lazily enables significant speedups in start-up time for applications. The mechanism consists in performing a very quick pass over relocations that can be resolved lazily (something that can be determined by the linker), setting them up such that, only when they are used for the first time are they actually resolved.

This has traditionally been used to resolve function addresses in dynamic linking. A call to a function that does not bind locally (i.e., that may be resolved to a definition in a separate module) is directed to go through a PLT entry, that loads an address from the GOT and jumps to it.

In the first pass, the dynamic loader sets these GOT entries to point to a stub that calls the dynamic resolver with enough information for it to identify the relocation that it should resolve at that time.

The dynamic resolver applies the relocation, modifying the GOT entry such that subsequent calls go straight to the actual function, and then transfers control to the function that should have been called, as if it had been called directly.

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.

In our optimized dynamic access model, however, there is a control transfer, and we realized we could use that to enable lazy relocation processing. In the quick pre-relocation pass, the function address in the TLS descriptor is set to another specialization that handles lazy relocation, and the argument is set so as to point to the relocation itself.

When the function is called, it resolves the symbol the relocation refers to, decides whether to use the Static or Dynamic specialization and sets up the TLS descriptor according to the decision, such that subsequent calls involving the same TLS descriptors go straight to the most efficient specialization.

Care must be taken to ensure that the TLS descriptor is never in a state that, should another thread perform an access using it, will yield an incorrect result.

On FR-V, that is not very difficult, since the instructions that read and store a pair of words are atomic given sufficient alignment. On IA32 and AMD64/EM64T, however, there is no instruction that can read or modify a pair of words atomically. Since requiring every call site to use synchronization would be too costly, a solution was devised that requires synchronization only in the lazy relocation function itself.

The lazy relocation specialization first acquires a dynamic loader lock and verifies that the TLS descriptor still points to itself. If so, it modifies it so as to point to a hold function and reads the argument. At that point, it can release the lock and compute the final value of the TLS descriptor, using the argument read while the lock was held.

Before modifying the descriptor, it acquires the lock again, wakes up any threads that might be waiting for it in the hold function (using say a condition variable), finally releasing the lock and transferring control to the function whose address was stored in the TLS descriptor.

The hold function simply acquires the lock and, in a loop, tests whether the TLS descriptor still points to it and, if so, waits on the condition variable until it is signaled, otherwise, it releases the lock and transfers control to the function specified in the TLS descriptor.

A simpler, yet less scalable, alternate design for the hold function, that does not involve condition variables, relies on the lock alone: the lazy relocation function does not release the lock throughout its operation, and the hold function is as simple as acquiring the lock, releasing it and transferring control to the function specified in the TLS descriptor. This design is quite appropriate when the relocation-processing code in the dynamic loader already requires a lock to be held, as it is the case in GNU libc.

4 Performance

Verifying any actual performance improvements provided by the optimizations introduced herein proved to be a major challenge. To the best of our knowledge, the only library that makes heavy use of Thread Local Storage is GNU libc itself. To make matters worse, GNU libc takes advantage of the fact that its dynamic loader and C library are always loaded initially, and thus they use the Initial Exec access model throughout the libraries offered by GNU libc, ensuring that any thread-local variables accessed with this access model are located in one of these two libraries.

Even forcing GNU libc to not use the Initial Exec access model and running the Native Posix Thread Library (NPTL[14]) performance benchmark to evaluate the benefit of the optimization to this benchmark showed no difference whatsoever. Investigation showed that this benchmark called `__tls_get_addr` only a handful of times during a test run that took tens of seconds, so performance differences could not possibly be exposed by this benchmark.

The main reason as to why the thread performance test did not use dynamic access models very often is that, first of all, it did not exercise thread-local storage access itself and, even if it did, it is a main application, not a dynamic library, so dynamic models do not apply. As for the libraries it uses, GNU libc's C and thread libraries maintain information pertaining to threads in the thread's static TLS block, and access it using a model similar to Local Exec, so they are not affected by the choice to not use the Initial Exec model within libc.

Although Gomp[15], the implementation of OpenMP[16] for the GNU toolchain, has very recently become a viable platform for measuring TLS performance, the SPEC OMP2001 benchmark uses `threadprivate` variables in only one of its tests, and even then, not in a dynamic library, so using this benchmark was not viable either, and we were left with the need for creating synthetic microbenchmarks.

We have created a total of 40 tests for our benchmark, such that every test is represented as a function that returns a result that is somehow related with one or more thread-local variables, with variations in 4 different dimensions, described in the following paragraphs.

Operation Half of the tests compute the address of a thread-local variable (**addr**), whereas the other half computes the actual value stored in the thread-local copy of the variable (**load**). This exposes differences related with the efficiency of accessing a thread-local variable without explicitly adding the thread pointer to its relative location. On all tested CPUs, the TP register is a special register whose contents cannot be read or modified from userland. It can be used as a base register to read or modify a thread-local variable, but computing the address of a variable requires loading the register's value from a reserved location in the Static TLS block.

Timing All of the timing is performed using the clock tick counting instructions available on the CPUs we've used for testing. Half of the test functions time their operation by themselves (**Internal**), storing the number of clock ticks elapsed while performing the operation in a pointer passed in as an argument. The other half perform no timing whatsoever, relying on their callers to obtain the clock tick count for the entire call (**External**). Unlike the previous dimension, that intends to expose differences, this one intends to confirm the performance improvements we've achieved, by offering multiple performance measures of different but functionally-similar code.

The confirmation was not straightforward, though; the little room for scheduling in the internal timing variants and the high pressure on the registers used by both the timing instructions and function call return values would create pipeline bubbles that, without care to avoid such worst-case conditions (unlikely to occur in real life), would have made some tests that perform very little work appear to be slower than some that perform much more work.

Access model We have four different kinds of tests in this dimension, in which knowledge about the location of the thread-local variable used varies, plus one kind of test that combines access to multiple variables.

Half of the single-variable tests use Initial Exec access models, but in half of these, the compiler generated Initial Exec code because it was told the variable was in Static TLS (**OIE**, for original IE); in the other half, the compiler was told the variable was in Dynamic TLS, so it generated General Dynamic code, and then the linker relaxed that to Initial Exec, being aware of the Static TLS location of the variable (**RIE**, for relaxed IE).

The other half of the single-variable tests use General Dynamic access models. In half of these, the variable is in Static TLS, so our main optimization kicks in (**SGD**, for static GD); in the other half, the variable is in Dynamic TLS, so the main optimization does not apply (**DGD**, for dynamic GD).

The multi-variable tests (**Cmb**, for combined) subtract the values or addresses of the **RIE** and the **SGD** variables, and adds the value or address of the **DGD** variable, returning the result. All this work grants the compiler more opportunity to hide the latency of certain operations through instruction scheduling.

Local State Half of the test functions are so simple that, when they have to call `__tls_get_addr` or equivalent, any automatic variables of their own can easily be assigned to call-preserved registers, so the optimized calling conventions suggested in this paper show no benefit whatsoever (**Min St**). In order to expose such benefits, the other half of the test functions contain a large number of automatic variables (**Max St**) whose contents are forced into registers before and after the TLS operation, such that, with the standard calling conventions, almost all call-clobbered registers have to be spilled before the call and reloaded after it, whereas with our optimization, none of this takes place.

The number of variables is chosen such that all but one of the general-purpose registers are taken up by these variables. On IA32, we use 5 such variables, considering that `%ebx` is reserved as the GOT pointer, and that `%ebp` can be used as a general-purpose register, making up for 6 available registers, 3 call-saved, 3 call-clobbered. On AMD64, we use 14 such variables, since `%esp` is not really usable in the 16-register set. On both CPUs, we keep one register available to hold the result of the TLS operation, with the explicit intention of showing a worst-case scenario for the traditional code, where the advantages of the custom calling conventions would be greatest. The actual benefit from this change will be somewhere in between the two variants in this dimension.

The 40 combinations of the above variations are all located in a dynamic library that is dlopened by the main benchmark program. This ensures that the test functions do not get inlined into the main benchmark loop, which might enable hoisting of operations, making operations look faster than they are.

We build two such dynamic libraries for each tested architecture: one created with the compiler configured to generate code in the traditional way (**Ol**), another following our new proposed method (**Nu**). A full test run goes through all 40 tests for each of the 2 libraries, which makes up for the 80 tests total.

Every test is run a large number of times, in two different configurations. In one configuration, we run each one of them in a tight loop to then proceed to the next test; in the other, each test is run once in a randomized sequence generated for every iteration in a loop. More details are given below.

Although running the tests in a tight loop has enabled us to initially measure a lower bound for the execution time of each test, such lower bound was initially not thought to be very representative of real-life performance, since it depends heavily on hot caches and nearly-perfect branch and call/return prediction, something that is not necessarily expected in practice.

In order to try to obtain more representative results, we collect all of the tests into a vector and then, for every iteration in the main benchmark loop, we get the vector sorted at random and then iterate over the randomized vector, running each test once per iteration in the main loop. Each test run produces a time result that is immediately logged to a file. This logging and randomization helps avoid getting cache, branch and call/return prediction hits too common for any single test, which enables us to achieve moderately reproducible results with thousands of runs of each test, as opposed to hundreds of millions that we needed in the tight-loop test. It often (but not always) gets us identical per-iteration lower bounds, but the average run times no longer tend to the lower bound as the iteration count increases.

Unfortunately, this randomization, and the possibility of long interrupts and context switches that could skew averages up at random, have caused average times over 1 million runs to vary by as much as 30%, even after discarding values that appear to be too high.

That said, in spite of the significant error margin in the exact averages, we've verified that there appears to be a strong correlation between improvements in minimum times, as measured in the tight loop, and improvements in the average times, although speedups tend to be smaller for averages than for minimums.

Given this correlation and the irreproducibility of the exact average results, we've decided to not include the average times in the paper. Since binaries and the complete source code of the implementation, including the benchmark program that can generated them, are available for download at `http://www.lsd.ic.unicamp.br/~oliva/`, publishing only the minimum times, that are perfectly reproducible, was deemed enough.

4.1 Analysis

Testing procedure was as follows. A toolchain was built on Fedora Core 4, based on snapshots of the GCC and GNU binutils development trees taken on Oct 30, 2005. This toolchain was capable of generating code for both IA-32 and AMD64, selecting the old or the new TLS call sequences through a command-line switch. A development snapshot of GNU libc, taken on the same day, was built using this compiler for both IA-32 and AMD64. The IA-32 version was built with optimizations for Pentium II or newer; the AMD64 version was built with default settings. The benchmark program and libraries were built with the same settings.

The benchmark program was run on 3 different environments, each one described in the caption of the corresponding table: a Pentium III processor ran the 32-bit benchmark (Table 1),

		Internal Timing				External Timing			
Acc		Min St		Max St		Min St		Max St	
Mod	Op	Ol	Nu	Ol	Nu	Ol	Nu	Ol	Nu
OIE	load	33	33	37	37	48	48	58	58
	addr	33	33	38	38	45	45	55	55
RIE	load	35	35	40	38	50	50	61	60
	addr	34	34	39	37	48	50	62	59
SGD	load	64	39	67	43	77	53	88	64
	addr	63	39	67	43	76	53	87	64
DGD	load	64	58	67	58	77	67	90	78
	addr	63	53	68	58	76	67	87	76
Cmb	load	104	63	108	77	110	78	131	100
	addr	94	64	101	69	113	78	123	90

Table 1: Minimum run times, in CPU cycles, over 100000000 iterations on a Pentium III Speedstep 1.0GHz (32-bit only). The timing overhead, included in the figures above, was measured as 33 CPU cycles.

and an Athlon64 processor ran both the 32-bit (Table 2) and the 64-bit (Table 3) benchmarks. In all cases, the processors were configured to avoid clock speed switching, and the boxes were very lightly loaded, except for the benchmark program. The results were mechanically converted to LaTeX tables.

Figures 3, 4, 5, and 6, also generated mechanically, display information from the **SGD** and **DGD internal-timing** tests in the tables. In each chart, the left cluster of bars is for **Min St** tests; that on the right is for **Max St** tests. Within each cluster, the bars represent each of the tested machines, in the same order that their tables appear. Within each bar, the dotted line represents the timing overhead (see below), the lower bar is the **Nu** time and the upper bar is the **Ol** time. Speedups are computed in each bar; the lower speedup is computed as a fraction of the **Ol** and **Nu** numbers directly from the table, the upper speedup is computed by first subtracting the timing overhead from the dividend and the divisor. The real speedup in practice ought to be between the two figures.

The timing overhead is the difference in the clock tick count between two subsequent ex-

		Internal Timing				External Timing			
Acc		Min St		Max St		Min St		Max St	
Mod	Op	Ol	Nu	Ol	Nu	Ol	Nu	Ol	Nu
OIE	load	9	9	10	10	24	24	29	29
	addr	5	5	10	10	21	20	30	29
RIE	load	9	9	17	10	25	25	34	30
	addr	5	5	13	10	21	22	30	29
SGD	load	34	9	40	15	49	29	57	31
	addr	32	9	38	11	44	25	56	31
DGD	load	35	23	40	25	48	38	57	42
	addr	31	18	38	21	46	37	56	40
Cmb	load	76	29	79	39	78	46	98	59
	addr	66	23	68	32	76	42	87	49

Table 2: Minimum run times, in CPU cycles, over 100000000 iterations on an Athlon64 3000+ (1.8GHz) notebook, running the benchmark compiled for 32-bit mode. The timing overhead, included in the figures above, was measured as 8 CPU cycles.

		Internal Timing				External Timing			
Acc		Min St		Max St		Min St		Max St	
Mod	Op	Ol	Nu	Ol	Nu	Ol	Nu	Ol	Nu
OIE	load	9	9	9	9	9	9	19	19
	addr	8	8	8	8	9	10	18	17
RIE	load	9	9	22	9	13	9	32	19
	addr	5	8	20	8	9	10	28	16
SGD	load	26	9	37	11	29	15	47	20
	addr	23	9	36	10	28	12	47	18
DGD	load	26	25	37	25	29	26	48	31
	addr	23	21	36	21	28	22	47	31
Cmb	load	47	30	62	39	52	31	72	50
	addr	42	23	59	28	49	27	68	37

Table 3: Minimum run times, in CPU cycles, over 100000000 iterations on the same Athlon64 notebook from Table 2, running the benchmark compiled for 64-bit (AMD64) mode. The timing overhead, included in the figures above, was measured as 5 CPU cycles.

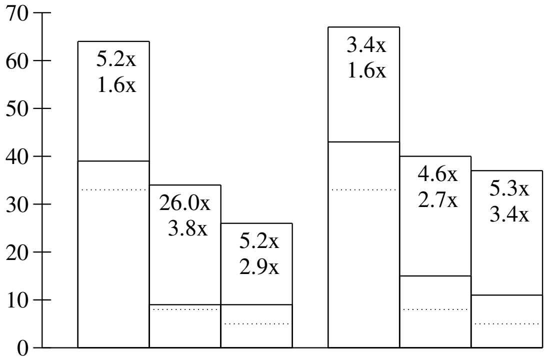

Figure 3: SGD load internal timing results.

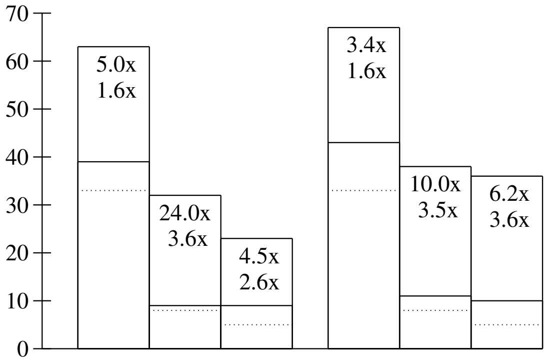

Figure 4: SGD addr internal timing results.

ecutions of the instruction that obtains this count, including the time needed to copy the contents of the first measurement elsewhere before they are overwritten by the second measurement. Careful analysis of the tables shows that the overhead is greater than or equal to the times measured for certain simple operations. Such simple instruction sequences are believed to fit in, or even help avoid additional pipeline bubbles.

OIE tests confirm the expected absence of variation, given that it is the exact same code being generated for both the old and the new TLS conventions.

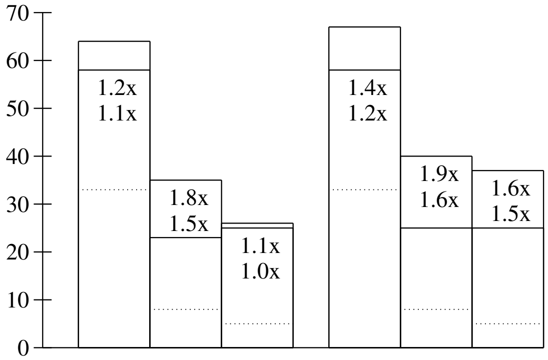

Figure 5: DGD load internal timing results.

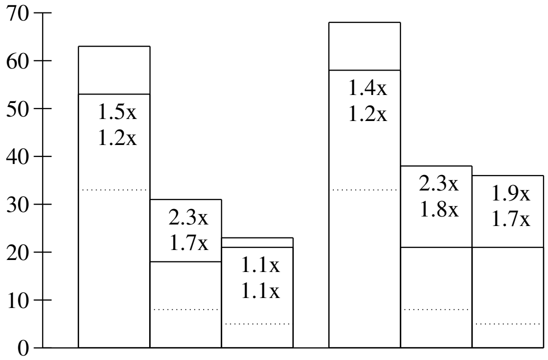

Figure 6: DGD addr internal timing results.

RIE remains nearly identical in terms of performance on IA-32 for the minimum-state tests, as expected. For the maximum-state tests, the new method begins to show improvements, as it enables the compiler to preserve more state across the TLS calls that, in these tests, end up being relaxed, but the advantage remains since the linker cannot recover the performance loss due to register spilling and reloading. The performance loss in the 64-bit minimum-state address RIE probably indicates there might be better instruction sequences we could use for relaxation.

SGD is where the new method really shines. That is no surprise, since it's exactly the situation that the new method is designed to improve, and fortunately also the most common situation in code generated for dynamic libraries that accesses thread-local variables. Absolute reductions in clock cycles are consistent between internal and external timing in 32-bit mode, where the calling conventions optimization plays a less significant role; in 64-bit mode, the absolute reductions in clock cycles are consistent if you compare results among the minimum-state tests, or among the maximum-state ones.

DGD shows that performance is improved significantly even in the case that the new method regarded as the slow case. Clearly, in 64-bit mode, most of the savings stem from the optimized calling conventions, that enable the retention of state in registers, as shown in the comparison between minimum- and maximum-state in the internal timing column, where the new model remains unchanged upon the growth in state and the old model slowed down by a significant amount. In the external timing column, the overhead from having to preserve all callee-saved registers that are used is noticeable in the maximum-state column, but not as much as in the old model. In 32-bit mode, the ability to check whether the DTV is up-to-date without setting up the GOT pointer is likely what brings most of the benefit.

Cmb essentially only confirms the results above, not offering any obvious new insights.

5 Conclusion

The proposed optimization improves performance of access to thread-local variables from dynamic libraries by a big margin for initial libraries, without any data size penalty and most often with code size reductions. For dlopened libraries, there are still performance advantages, but to a lesser, yet still significant extent, and there are data size penalties.

It should be highlighted that the performance gains from lazy relocations, by avoiding relocation processing at load time, and from code size reductions, by improving the instruction cache hit rate, have not been taken into account at all in the micro-benchmarks exposed here.

The implementation is readily available for widely-used CPU types, under Free Software licenses that enable any library to take advantage of this novel technique.

Some open questions remain to be answered in future work: whether there are relaxation sequences that could make the new relaxed code at least as fast as the old one on AMD64, and faster on IA32; whether returning an offset instead of an address from the specialized `__tls_get_addr` calls does indeed help improve performance; whether enabling the specializations to clobber one or two registers, which would enable the dynamic-case fast path to save fewer or even no registers, would cause a measurable decrease in performance in the more common cases; how much of a performance improvement could have been obtained over the old model by using the same call sequences, and only modifying the run-time so as to compute relocations differently, and modifying `__tls_get_addr` to cope; how much benefit would be obtained by implementing DTV compression; how well the optimizations described here do on other architectures.

Acknowledgements

Alexandre Oliva thanks Glauber de Oliveira Costa, for having reviewed this paper and offered to extend this work to the ARM platform; colleagues Roland McGrath, Jakub Jelínek, Richard Henderson, and Ulrich Drepper for the early discussions on the design, for the support and reviews in designing and implementing the optimizations on various architectures; Aldy Hernandez for tolerating the delays due to the creativity in designing the FR-V TLS ABI, for drafting its early descriptions and for doing the compiler work for that implementation; Eric Bachalo, his manager at that time, for giving his approval to such creativity in a customer-funded project, approval that in the end made this work possible.

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] 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.

[8] 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.

[9] Fred C. Chow. Minimizing register usage penalty at procedure calls. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 85–94. ACM Press, 1988.

[10] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[11] Kevin Buettner, Alexandre Oliva, and Richard Henderson. The FR-V FDPIC ABI. `http://people.redhat.com/aoliva/writeups/FR-V/FDPIC-ABI.txt`, April 2004. Version 1.0.

[12] Intel Itanium Processor-specific Application Binary Interface (ABI). `http://refspecs.freestandards.org/elf/IA64-SysV-psABI.pdf`, May 2001.

[13] Ian Lance Taylor. 64-bit PowerPC ELF Application Binary Interface Supplement. `http://www.linuxbase.org/spec/ELF/ppc64/PPC-elf64abi-1.7.pdf`, September 2003. 1.7 Edition.

[14] Ulrich Drepper and Ingo Molnar. The Native POSIX Thread Library for Linux. `http://people.redhat.com/drepper/nptl-design.pdf`, February 2005.

[15] GOMP — An OpenMP implementation for GCC. `http://gcc.gnu.org/projects/gomp/`, November 2005.

[16] OpenMP Architecture Review Board. OpenMP Application Programming Interface. `http://www.openmp.org/drupal/mp-documents/spec25.pdf`, May 2005. Version 2.5.