Featherweight Threads and ANDF Compilation of Concurrency*

Ben Sloman[1,2] and Tom Lake[2] **

[1] University of Reading, Department of Computer Science, PO Box 225, Whiteknights, UK, RG6 2AY
[2] GLOSSA, 59 Alexandra Road, Reading, RG1 5PG, UK

Abstract. We present an intermediate representation called ThreadTDF, a component of the Parallel TDF system for compiling distributed concurrent programs to shared and distributed memory multiprocessors. ThreadTDF is a parallel extension of the TDF architecture neutral distribution format (ANDF) for sequential programs. ThreadTDF provides *featherweight* thread mechanisms for explicitly scheduling dynamic fine-grain concurrent computations within procedures (and more generally within static local scopes). Communication between address spaces is supported by remote service request mechanisms based upon asynchronous activation of remote threads and synchronous remote procedure calls. In ThreadTDF variable lifetimes bound the lifetimes of featherweight threads declared in their scope. We show how a compiler uses thread lifetime information to integrate resource allocation and communication with thread scheduling for efficient intraprocedural concurrency. Initial performance results are given for the SPARC processor.

1 Introduction and Background

Parallel TDF is a system for the architecture neutral representation and compilation of parallel programs. It consists of a family of architecture neutral compiler intermediate representations (IRs), of which ParTDF and ThreadTDF are currently the most important, and techniques for compiling parallel programs using these representations. Parallel TDF is based upon TDF[7], an architecture neutral and language neutral distribution format. TDF is a compiler IR with a standardised external representation. In one scenario, TDF is used to bridge language-specific compiler front ends (*producers*) and separate target-specific back ends (*installers*) for distribution of 'shrink-wrapped' software (Fig. 1). TDF cannot create portability[3]: applications must use portable programming styles and must adhere to standard APIs. Parallel TDF attempts to extend the functionality of TDF for parallel languages and machines.

TDF is described in detail in [7] and other information is available from the OSF ANDF web page at *http://riwww.osf.org:8001/andf/index.html* or in [3, 6].

* This work was carried out under contract for the UK Defence Research Agency
** The authors can be contacted as {Ben.Sloman,Tom.Lake}@glossa.co.uk.
[3] Though the TDF technology has proved useful for portability *checking* tools.

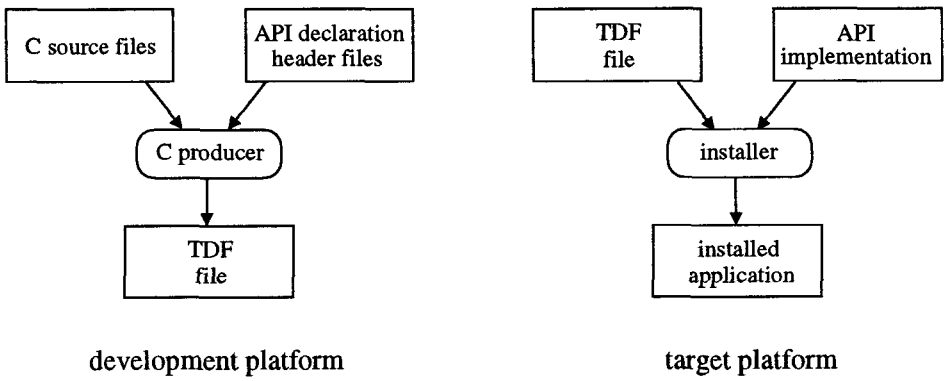

Fig. 1. shrink wrapped distribution using TDF

Parallel TDF provides services for concurrent computation and communication. Its design supports compilation of data parallel and control parallel languages on SIMD and homogeneous shared and distributed memory MIMD machines. Structured parallel languages such as Fortran 90 and occam are compiled into the ParTDF intermediate representation in terms of bags (multisets) of processes. ParTDF allows portability-improving transformations such as distribution of bulk parallelism and conversion of task parallelism for lockstep parallel execution [15]. Other parallel languages are compiled into the ThreadTDF intermediate representation. ThreadTDF provides dynamic manipulation of scoped fine grain threads for efficient concurrency on a narrower range of architectures.

Our contribution is to describe ThreadTDF and show how it can be compiled efficiently to off-the-shelf processor architectures. We show that thread scopes simplify register allocation across threads. We then show how to exploit the register allocation by using cooperative hierarchical scheduling to optimise context switches between threads in the same variable scopes.

2 Introduction to TDF

TDF is a tree-structured language with special features for portability. It preserves more program structure than low level IRs such as the RTL of *gcc*, but has no syntactic sugar and a weaker type system than typical high level languages. The unit of representation in TDF is the *capsule* which contains definitions and declarations of procedures, variables and tokens (see below). A capsule can export these declarations and definitions by binding them to external names. The ANSI C TDF producer converts each separate C file into a compact binary representation of a TDF capsule. A TDF linker allows capsules to be bound together using their external names.

In TDF, any target machine or OS dependences, such as the implementa-

tions of C types **int** and **FILE**, are deferred using parameterised placeholders, similar to syntax macros, called *tokens*. The TDF linker can be used to bind in token definitions once the target machine is known. The resulting TDF is then converted to object code by an installer and linked using the system linker to produce an executable (see Fig. 1).

TDF producers have been or are being developed for C, Fortran 77, Ada, Dylan and C++ along with installers for SPARC, 80x86, MIPS, Alpha, PowerPC, HP/PA and ARM processors.

2.1 TDF's Tree Structure

The tree structure of TDF is defined as a multi-sorted abstract algebra. Each *sort* can be understood roughly as implementing a particular class of high level language construct. The most important TDF sorts are EXP, which represents executable operations (such as commands and expressions), and SHAPE which describes the types of the static-sized run-time values delivered and manipulated by EXPs. Each sort has a set of *constructors* (operators) each of which takes operands of known sort. Figure 2 specifies some of the EXP constructors. EXPs are annotated with their SHAPEs; there are SHAPEs for integer and floating point numbers, pointers, store offsets, and compounds of these values. The special shapes **TOP** and **BOTTOM** are used for constructs that return no usable value or that transfer control respectively.

```
plus (ERROR_TREATMENT,EXP INTEGER(v),EXP INTEGER(v)) -> EXP INTEGER(v)
make_int (v:VARIETY,SIGNED_NAT) -> EXP INTEGER(v)
assign (EXP POINTER(x),EXP(y)) -> EXP TOP
contents (s:SHAPE,EXP POINTER(x)) -> EXP s
integer_test (NTEST,LABEL,EXP INTEGER(v),EXP INTEGER(v)) -> EXP TOP
goto (LABEL) -> EXP BOTTOM
conditional (LABEL,EXP x,EXP z) -> EXP (x ⊔ z)
repeat (LABEL,EXP x,EXP z) -> EXP z
conditional (LABEL,EXP x,EXP z) -> EXP (x ⊔ z)
labelled (LIST(LABEL),EXP x,LIST(EXP)) -> EXP w
sequence (LIST(EXP),EXP x) -> EXP x
variable (OPTION(ACCESS),TAG,EXP x,EXP y) -> EXP y
identify (OPTION(ACCESS),TAG,EXP x,EXP y) -> EXP y
```

Fig. 2. some EXP constructors

The meaning of terms in the TDF algebra is specified in terms of the interleaved 'expression-like' evaluation of EXP constructors. Additional ordering may be imposed by declaration (initialiser before scope), sequence and by explicit jumps to LABELs. Values are delivered from operands to operators or passed via store using assignment and dereference.

The TDF specification [7] defines 53 different SORTS which are combined to produce an EXP for each procedure in a source language program. These EXPs

are combined with SORTS describing linking information and global declarations to produce a TDF capsule.

2.2 Declarations and Ordering

Values may be bound to unique numeric identifiers called TAGs. `obtain_tag` delivers the value bound to its operand TAG. `identify` binds the result of evaluating its first EXP operand to a TAG that may then be used during evaluation of its second EXP operand. `variable` is similar but store is allocated to contain the initialising value and the TAG is bound to the address of this store. Declarations are also provided for global TAGs.

A few other EXP constructors also evaluate their EXP operands in order. The simplest is `sequence` which evaluates its operands from left to right and delivers the last operand's result. More general ordering is provided by explicit jumps such as `goto` and `integer_test`; target LABELs are scoped by the declarations `conditional`, `repeat` and `labelled`. `conditional(LB, XA, XB)` evaluates `XA` with LABEL `LB` available for forward jumps to `XB`. `repeat(LB, XA, XB)` evaluates `XA` then `XB` and `LB` is available in `XB` for backward jumps to the start of `XB`. Finally, `labelled(XA, LBi, Xi)` declares a list of labels `LBi` and a list of corresponding target EXPs, `Xi`. Evaluation starts with `XA` and any label `LBi` may be used in any EXP operand.

3 ThreadTDF

ThreadTDF extends TDF in a natural way with mechanisms to dynamically create and schedule fine-grain concurrent EXP evaluations that we call *featherweight* threads. These mechanisms are general enough to express a wide range of concurrency: they provide much of the functionality of existing thread libraries, such as POSIX threads, but allow concurrency within as well as between procedure instances. Featherweight threads execute within a shared address space by default, communication between address spaces is added using a notion of remote thread activation.

The execution of multiple featherweight threads within a procedure instance provides new opportunities for integrating resource allocation with scheduling. It often becomes possible to allocate machine resources (stack and registers) statically across threads. An implementation using hierarchical scheduling, described in Sec. 4, groups thread executions by procedure instance to exploit static allocation within the procedure.

Featherweight threads are self-scheduled: there is no notion of a thread handle other than the thread value representing a descheduled thread. We provide lightweight mutual exclusion between threads: other synchronisation operations can be constructed from exclusion and scheduling. More complex services, such as `kill_thread` or priority control, are implemented using self-scheduling. This approach was first proposed in [8] while considering TDF extensions to support compilation of the concurrent object language UC++.

A key aim of ThreadTDF is to allow efficient exploitation of uniprocessors. This is important for many reasons. It allows parallel programs to ride the rapid 'technology curve' of sequential hardware improvements. It supports multiprocessing. It also eases the software development process and encourages portability and scalability.

3.1 Featherweight Thread Operations

A featherweight thread is a concurrent execution of an EXP starting at a label. Thread label declarations have a *Single Entry Single Exit* (SESE) property: a thread is only allowed to complete (fall through) a declaration once all other threads enclosed by the same instance of the declaration have terminated. This means that the lifetime of many threads is bounded by the lifetime of their initial labels. We currently require that threads synchronise explicitly to enforce the SESE property, thereby putting the onus on the ThreadTDF producer. We are also considering adding thread declarations that provide the required thread synchronisations implicitly. Section 4 shows how the SESE property helps code generation.

Figure 3 contains the ThreadTDF constructors. These include a new shape called THREAD for values representing descheduled threads and a new sort called THDLB for thread labels. Modified forms of `conditional` and `labelled`, called `par_conditional` and `par_labelled`, are used to declare thread labels. Procedure return and jumps to external labels are forbidden within these constructs. An instance of a thread label is *replicated* if more than one thread is created or suspended at it or yields to it during its lifetime. Only `par_labelled` labels can be replicated and each of these labels is associated with an optional bound on its degree of replication.

A descheduled thread is created suspended at a THDLB by `create_thread`. An executing thread suspends itself at a THDLB using `suspend_thread`; execution continues in a new thread into which the suspended thread is delivered. Both creation and suspension are parameterised by pointers that address space at which a new thread's internal values can be stored. This space must be allocated explicitly by `variable` or by dynamic store allocation using the size delivered by `thread_size` (parameterised by the amount of store required for user thread-local values). A thread value can be scheduled for eventual execution using `schedule_thread`, and execution is terminated using `stop_thread` or `swap_thread`. Regular use of `yield_thread` is required so that cooperative scheduling can ensure fair independent progress.

A simple extension of thread creation and suspension provides creation and suspension at thread labels declared by previous procedure instances in the current procedure call chain. This can be used to create threads with unlimited lifetimes.

In some cases the new thread created at `suspend_thread` will never block[4] or call or return. In this case `nb_thread` can be used to supply space for the

[4] The blocking operations are: `mutex`, `suspend_thread` and `yield_thread`.

```
THREAD -> SHAPE
par_conditional (THDLB,OPT(EXP INTEGER(v)),EXP a,EXP b) -> EXP TOP
par_labelled (
  EXP a,LIST(OPT(EXP INTEGER(v))),LIST(THDLB),LIST(EXP b)) -> EXP TOP
create_thread (EXP POINTER,lb:THDLB) -> EXP THREAD
schedule_thread (EXP THREAD) -> EXP TOP
yield_thread (THDLB) -> EXP TOP
stop_thread -> EXP BOTTOM
suspend_thread (EXP POINTER,THDLB) -> EXP THREAD
thread_size (EXP OFFSET) -> EXP OFFSET
nb_thread -> EXP POINTER
swap_thread (THREAD) -> EXP BOTTOM
nbstop_thread -> EXP BOTTOM
nbswap_thread (THREAD) -> EXP BOTTOM
current_thread (EXP THREAD) ->
access_threadstore (EXP THREAD) -> EXP POINTER
make_null_thread -> EXP THREAD
test_thread (NTEST,LABEL,EXP THREAD,EXP THREAD) -> EXP TOP
KEY -> SHAPE
make_key -> EXP KEY
mutex (EXP POINTER,EXP b) -> EXP TOP
```

Fig. 3. basic thread operators

thread so long as it terminates using **nbstop_thread** or **nbswap_thread**.

Featherweight threads may synchronise using lightweight mutual exclusion. **mutex** takes a pointer to a key and executes its body EXP in mutual exclusion with all other bodies guarded by the same pointer. The body will not contain loops, thread label declarations, procedure calls or returns, or any other construct that may involve a context switch. This means that mutual exclusion comes for free when using cooperative scheduling on a uniprocessor.

ThreadTDF's 'continuation-passing' style of scheduling allows orthogonal combination of scheduling and mutex so, for example, threads can suspend into shared data structures under mutual exclusion. Thread termination within a mutex body or jumps to external labels cause the mutex to be released. Mutexes may be nested but in a correct TDF program the mutex pointer will always be strictly less, in some partial order, than that of any enclosing mutexes to prevent mutex deadlock.

3.2 Distributed Featherweight Thread Operations

We provide three mechanisms for communicating between address spaces. The simplest uses shared global TAGs to communicate static values: the runtime system implements any necessary communication. This mechanism is mainly used to bootstrap other forms of communication during program initialisation.

More complex communication mechanisms use remote values of shape RE-

MOTE. A value is converted to a remote by applying **make_remote**: the resulting value contains the original value, and the identity of the current processor (often an integer). The original value can be extracted from a remote by applying **localise** to it on the processor on which it was originally made.

A remote procedure value is used by **apply_proc_remote** which performs a blocking remote procedure call to the procedure on the processor on which the remote was created.

The *inlet* mechanism allows remote activation of a thread in an existing procedure instance on the target. An inlet is a form of featherweight thread with provision for arguments (similar to TAM inlets [4]). Figure 4 contains the inlet constructors. **make_inlets** declares a group of inlets and delivers their common 'environment'. Each inlet has a THDLB label, a list of formal parameter TAGs and shapes, and a body. **send** and **remote_send** allow asynchronous activation of an inlet using its label and environment. The list of inlet arguments supplied at the send must agree with the formals' declared shapes.

```
INLETENV -> SHAPE
make_inlets (LIST(THDLB),LIST(LIST(TAGSH)),LIST(EXP)) -> EXP INLETENV
send (EXP INLETENV,THDLB,LIST(EXP)) -> EXP TOP
remote_send (EXP REMOTE INLETENV,THDLB,LIST(EXP)) -> EXP TOP
```

Fig. 4. inlet operators

Experiences with active messages [16] and TAM inlets [4] indicate it may be necessary to impose appropriate disciplines on inlet use e.g. to prevent deadlock in the network or to simplify buffer allocation.

4 Implementing ThreadTDF

We now describe implementation techniques for the ThreadTDF operations. We use a form of *hierarchical cooperative scheduling*: preemption could also be used but is more complex. We have used these implementation techniques in a ThreadTDF installer for the SPARC processor built using the existing TDF SPARC installer. The ThreadTDF installer acts as the code generator for a Parallel TDF system for compiling occam and for compiling our own dynamically threaded dialect of C. Section 4.2 gives initial performance results.

Our ThreadTDF installer implements the scheduling of featherweight threads explicitly in terms of operations on a bidirectional ring of bidirectional rings[5]: a global ring of procedure instances and a local ring of thread instances for each procedure instance (shown in Fig. 5). We perform register and store allocation of variables across all threads in a procedure so that we can make context switches

[5] The ring structure provides fairness, if fairness were not required a stack would do.

within a procedure (around a local ring) cheap. Context switches between procedures (around the global ring) are usually more expensive. The hierarchy could be extended to include further subrings for contexts within a procedure.

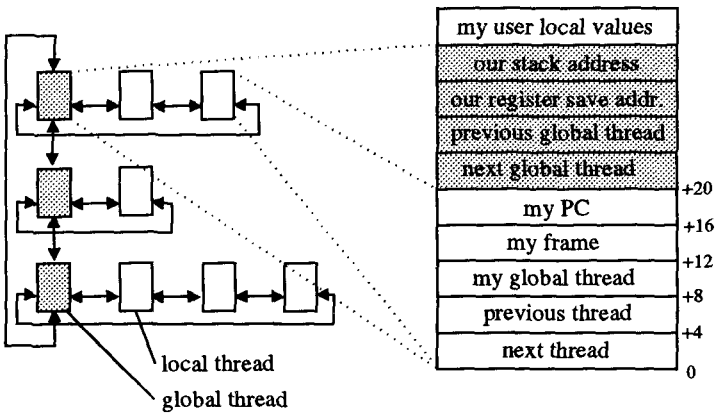

Fig. 5. data structures for hierarchical thread scheduling

A thread is scheduled by inserting it in its local ring but it only actually executes when this ring becomes current. This 'lazy' scheduling generates more local concurrency between global context switches and enhances locality[6].

4.1 Implementing Featherweight Threads

The first main task of the ThreadTDF installer is to provide dynamic control transfer for independent thread progress; the ring-of-rings data structure and thread data structure we use for this purpose are shown in 5. Below we describe how register allocation supports scheduling using these structures.

The second main task is to implement procedure local variables. These are allocated in registers, on a single common stack, and on the heap. The nesting of variables within thread declarations induces a tree of *frames*, each frame containing nested variables with the same degree of replication and local to the same threads. Frames are allocated using the most efficient applicable mechanism. The SESE structure of thread declarations improves this process as it ensures all lifetimes nest properly and this reduces the connectedness of the variable interference graph. This allows better reuse of registers and store locations which in turn improves access latency and spatial density and reduces dynamic allocation overhead. The SESE structure also ensures that the degree of replication nests properly.

[6] It is also possible to execute threads eagerly where advantageous e.g. when scheduling a thread in the current procedure instance that is known to run without blocking.

The third main task is to implement concurrent procedure calls. We wish to reuse the optimised sequential procedure calling convention so each concurrent call potentially requires a new system stack. The installer must also save and restore a procedure's local registers at global context switches and at procedure call. Callee-save registers are a particular issue (see below).

The final task of the installer is to implement as many standard sequential optimisations as possible in the presence of concurrency.

We will not discuss the implementation of inlets in detail here. An inlet is activated using active message mechanisms [16] but the target activation executes in a shared pre-existing procedure instance (see [4] for more detail).

Intraprocedural Concurrency. A THREAD value is simply a pointer to a region of memory containing the thread-local system and user values (shown in Fig. 5, shaded values are present only in global threads). We allocate variables to registers and store in such a way that only the program counter register needs to be saved for a thread. We do this by ensuring that register usage at context switch targets is a subset of the usage at context switch sources. In our implementation all context switches start at `yield_thread`, `stop_thread`, or `swap_thread` (or the non-blocking equivalents) and end at a thread label. The SESE property of thread declarations ensures that only variables in scope or declared in a concurrent operand of an enclosing thread declaration can be live at any point.

Each procedure maintains a pointer to the current local thread, usually in a register. Local context switch can be implemented in three machine operations: (1) load the next local thread's address (2) retrieve its PC (3) jump to it. This seems to work well for our applications because there are usually enough SPARC registers to hold the variables live at context switch targets. When we run out of registers we allocate the remaining variables to store but we could also provide thread-local register allocation if we were prepared to spill these registers at context switches.

Scheduling operations, such as `create_thread` and `schedule_thread` are implemented in terms of simple pointer operations on the scheduling data structures. Every thread is created containing the address of its global thread (this may need to be updated in called procedures) so it can always be scheduled by inserting it in the local ring at this thread. Non-blocking threads need no entry in the scheduling data structures so thread operations on non-blocking threads are often particularly simple.

Local Variables. A local variable is *atomic* if its scope contains no thread label declarations. Atomic variables can be allocated to register or stack using sequential techniques because they are never live at any context switch target. Unfortunately non-atomic variables require a more involved treatment.

A procedure's non-atomic local variables are allocated in registers and in a tree of frames spread across the stack and heap. Together these mechanisms provide the logical 'tree-of-frames' structure induced by the nesting of concur-

rency in variable scopes. The installer analyses the thread declaration structure and local variable declarations to construct the largest possible frames, and to determine the number of replicated instances if possible. Dynamically replicated frames are allocated on the heap, others on the stack. Dynamic allocation is performed on entering a thread declaration and released on exit. `create_thread` and `suspend_thread` operations initialise their threads' frame pointers, if required, using the next free frame for the given thread label (the continuation of a suspended thread inherits the suspended thread's frame). If there is no free frame then a new chunk of frames is allocated and added to a free list associated with the label.

Interprocedural Concurrency. Many ThreadTDF procedure calls are purely sequential and can be executed with a normal sequential procedure call. This includes calls outside thread declarations or when the local ring contains only one local thread. External procedures, those that are not generated by the installer, can also be treated as sequential (though not external calls that call back to internal procedures). Run-to-completion calls, using `apply_rtc_proc`, can also be treated as sequential in many cases. All other procedure calls are concurrent.

Concurrent procedure calls impose a (mainly) small overhead due to the need to manage concurrency. In the worst case a new global thread is allocated and a new stack for the call to execute upon. After this the call uses existing sequential mechanisms. Sometimes it may be possible for an installer to determine the size of stack to allocate at a call, otherwise we resort either to guessing, or to using stack check mechanisms. The stack size problem is particular difficult when calling external procedures, it is probably best to sequentialise these calls on a system stack so that the normal stack checking mechanisms apply. Further overhead may be required to tidy up callee-save registers when a call returns if it has been interrupted for continued execution of a thread in the calling procedure.

Figure 5 shows how a ring of global threads is constructed, one for each procedure instance, to allow independent progress of concurrent calls. A local context switch to the global thread causes, with compiler-controlled frequency, a global context switch to the next procedure instance in the global ring. This switch must save the outgoing thread's registers (and register windows on the SPARC) and restore the registers of the target procedure instance.

We are also experimenting with lazy allocation techniques that reduce concurrency overhead when execution is sequential. For example, procedure calls can execute on their caller's stack so long as the calling procedure is provided with a new stack if it is rescheduled concurrently with the call.

4.2 Measurements

Figure 6 describes the wall-clock durations of individual featherweight thread operations. Times were measured on a SPARCstation 10 model 30 in single user mode. For comparison [11] cites thread creation times in the range 400–1300μs and thread switch times between 21 and 81μs for thread packages (including the

Sun Lightweight Process library) on the SPARCstation 10. Notice that the duration of `thread_create` depends on the degree of replication of any non-atomic variables in the new thread. Dynamically replicated variables are allocated in chunks for many threads at once using `malloc`. Our figures show how the per-thread overhead due to `malloc` increases as chunk size decreases from 1000 thread instances to 1.

Operation	Duration (μs)	
empty call	0.25	empty sequential call
create thread	0.18	no replicated variables
create thread	0.44	statically replicated variables
create thread	0.50–1.90	dynamically replicated variables
schedule thread	0.18	
local yield	0.23	between threads in the same procedure instance
global yield	14.0	between concurrent caller and callee procedures
concurrent call	16.0	set up stack and scheduling structures at call

Fig. 6. thread performance on a Sun SPARCStation 10

Figure 7 compares performance of two simple occam programs compiled using ThreadTDF and using the Southampton Portable Occam Compiler (SPOC)[5]. SPOC compiles occam to C which was then compiled using tcc or gcc (`-O2`). tcc is the TDF C compiler from which the ThreadTDF installer is derived. *daxpy* is a simple numeric kernel performing a floating point add and multiply per iteration. The inner loop is sequential in the SPOC version and parallel in the ThreadTDF version but the parallelism is eliminated during compilation so the end results are identical. *comstime* passes a value around a ring of channels connecting four occam processes. Here the iteration time measures the cost of four complete channel synchronisations and some internal scheduling overhead.

Platform	Application	Compiler	Iteration Time (μs)
SS10	daxpy	SPOC and GNU gcc	0.30
SS10	daxpy	SPOC and tcc	0.35
SS10	daxpy	ThreadTDF	0.35
25MHz T800	comstime	Inmos occam	12.0
SS10	comstime	SPOC and GNU gcc	12.0
SS10	comstime	SPOC and tcc	23.0
SS10	comstime	ThreadTDF	5.0

Fig. 7. performance of compiled occam

5 Related Work and Conclusion

As far as we know, ThreadTDF is the first language- and machine-independent intermediate representation for compiling concurrency. However many parts of the problem have been considered before.

Much work on compiling parallel languages has considered bulk parallelism and SPMD computation[14, 1, 12]. We believe that ThreadTDF should provide a suitable implementation substrate for these languages, though many optimisations and distribution transformations will be performed outside ThreadTDF, perhaps using ParTDF[15].

Another common approach uses portable libraries, e.g. PVM or POSIX threads, to support the compilation of concurrency. The level of functionality provided by such libraries can impose significant performance costs[10, 13] particularly when the source language allows both communication and concurrency.

Newer libraries, such as MPI and Chant[11], support communication with concurrency. Some, such as Nexus [9], are even designed as compiler targets. Nexus lacks featherweight threads, but otherwise provides services similar to ThreadTDF. It also supports global pointers and heterogeneous machines which ThreadTDF does not, yet. Despite the continued evolution of libraries, we still expect an intermediate representation to offer better performance. Firstly, procedure calls add their own overhead: call and return on a SPARC costs about the same as a *featherweight* context switch. Secondly, procedure calls to libraries prevent sharing in registers. Finally, the structure of an intermediate representation can support many important techniques, such as featherweight threads, that would not otherwise be possible. Technologies based on TDF should be widely portable, though libraries may be more widely available, at least initially.

ThreadTDF is close in spirit to the Threaded Abstract Machine [4]. Inlets and featherweight threads are similar to TAM inlets, but are not bound to execute to completion. Unlike TAM, ThreadTDF preserves sequential control structure and also supports more general thread operations.

In future we hope to improve support for distribution, add further source languages and perhaps add support for restricted scheduling disciplines, such as those considered in [2]. Our implementation techniques could be improved in many areas including register allocation, and dynamic store management. Also outstanding are consideration of remote memory access, memory consistency and of heterogeneous processing.

Finally it is worth noting that the ThreadTDF implementation techniques, if applied widely, may have significant implications for processor design.

References

1. V. Bala and J. Ferrante. Explicit data placement(XDP): A methodology for explicit compile-time representation and optimisation of data movement. *ACM SIGPLAN notices*, 28(1):28–31, January 1993.
2. Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foun-*

dations of Computer Science (FOCS '94), pages 356–368, Santa Fe, New Mexico, November 1994.

3. Frédéric Broustaut, Christian Fabre, François de Ferrière, Éric Ivanov, and Mauro Fiorentini. Verification of ANDF components. In *Proceedings of the 1995 ACM Workshop on Intermediate Representations*, 1995.
4. Daved E. Culler, Seth Copen Goldstein, Klaus Erik Schauser, and Thorsten von Eicken. TAM - a compiler controlled threaded abstract machine. Report, Computer science division, University of California, 1993.
5. Mark Debbage, Mark Hill, Sean Wykes, and Denis Nicole. *Southampton's Portable Occam Compiler (SPOC): User Guide.* University of Southampton, Southampton, UK, March 1994.
6. Stephen L. Diamond and Gianluigi Castelli. Architecture Neutral Distribution Format (ANDF). *IEEE Micro*, 14(6):73–76, December 1994.
7. DRA. *TDF Specification, Issue 3.* Open Software Systems Group, St. Andrews Rd, Malvern, Worcs, WR14 3PS, UK, March 1994. Obtainable via WWW from http://riwww.osf.org:8001/andf/andf.papers/toc.html.
8. P. W. Edwards, D. I. Bruce, D. J. C. Hutchinson, I. F. Currie, and P. D. Hammond. TDF and parallel object oriented languages. Deliverable 2.2 IED3/1/1059, Defense Research Agency, Open Software Systems Group, St. Andrews Rd, Malvern, Worcs, WR14 3PS, UK, 1992. October.
9. Ian Foster, Carl Kesselman, and Steven Tuecke. Nexus: Runtime support for task-parallel programming languages. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1994.
10. E. W. Giering, Frank Mueller, and T. P. Baker. Implementing Ada 9X features using POSIX threads: Design issues (draft). Technical report, New York University, Computer Science Department, New York, NY 10003, USA, 1993.
11. Matthew Haines, David Cronk, and Piyush Mehrotra. On the design of Chant: A talking threads package. In *Proceedings of Supercomputing 94*, pages 350–359, November 1994. Washington, DC.
12. S. Hiranandani, K. Kennedy, and C. Tseng. Preliminary experiences with the Fortran D compiler. In *Supercomputing 93*, pages 338–350, November 1993. Portland, Oregon.
13. Wilson C. Hsieh, Kirk L. Johnson, M. Frans Kaashoek, Deborah A. Wallach, and William E. Weihl. Efficient implementation of high-level languages on user-level communication architectures. Technical report, MIT, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Massachusetts 02139, May 1994.
14. V. B. Muchnick and A. V. Shafarenko. F-code: A portable software platform for data-parallel lanaguages. Technical report, Dept. of Electronic and Electrical Engineering, University of Surrey, April 1992.
15. Ben Sloman and Tom Lake. Extending TDF for concurrency and distribution. Report, GLOSSA, 59 Alexandra Road, Reading, RG1 5PG, UK, January 1995.
16. T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, May 1992. Gold Coast, Australia.