# Pillar: A Parallel Implementation Language

Todd Anderson, Neal Glew, Peng Guo, Brian T. Lewis, Wei Liu, Zhanglin Liu, Leaf Petersen, Mohan Rajagopalan, James M. Stichnoth, Gansha Wu, and Dan Zhang

Microprocessor Technology Lab, Intel Corporation

**Abstract.** As parallelism in microprocessors becomes mainstream, new programming languages and environments are emerging to meet the challenges of parallel programming. To support research on these languages, we are developing a low-level language infrastructure called *Pillar* (derived from Parallel Implementation Language). Although Pillar programs are intended to be automatically generated from source programs in each parallel language, Pillar programs can also be written by expert programmers. The language is defined as a small set of extensions to C. As a result, Pillar is familiar to C programmers, but more importantly, it is practical to reuse an existing optimizing compiler like gcc [1] or Open64 [2] to implement a Pillar compiler.

Pillar's concurrency features include constructs for threading, synchronization, and explicit data-parallel operations. The threading constructs focus on creating new threads only when hardware resources are idle, and otherwise executing parallel work within existing threads, thus minimizing thread creation overhead. In addition to the usual synchronization constructs, Pillar includes transactional memory. Its sequential features include stack walking, second-class continuations, support for precise garbage collection, tail calls, and seamless integration of Pillar and legacy code. This paper describes the design and implementation of the Pillar software stack, including the language, compiler, runtime, and *high-level converters* (that translate high-level language programs into Pillar programs). It also reports on early experience with three high-level languages that target Pillar.

## 1 Introduction

Industry and academia are reacting to increasing levels of hardware concurrency in mainstream microprocessors with new languages that make parallel programming accessible to a wider range of programmers. Some of these languages are domain-specific while others are more general, but successful languages of either variety will share key features: language constructs that allow easy extraction of high levels of concurrency, a highly-scalable runtime that efficiently maps concurrency onto available hardware resources, a rich set of synchronization constructs like futures and transactions, and managed features from modern languages such as garbage collection and exceptions. In addition, these languages will demand good sequential performance from an optimizing compiler. Implementing such a language will require a sizable compiler and runtime, possibly millions of lines of code.

To reduce this burden and to encourage experimentation with parallel languages, we are developing a language infrastructure called Pillar (derived from Parallel Implementation Language). We believe that many key parts of the compilers and runtimes

for these languages will have strong similarities. Pillar factors out these similarities and provides a single set of components to ease the implementation and optimization of a compiler and its runtime for any parallel language. The core idea of Pillar is to define a low-level language and runtime that can be used to express the sequential and concurrency features of higher-level parallel languages. The Pillar infrastructure consists of three main components: the Pillar language, a Pillar compiler, and the Pillar runtime.

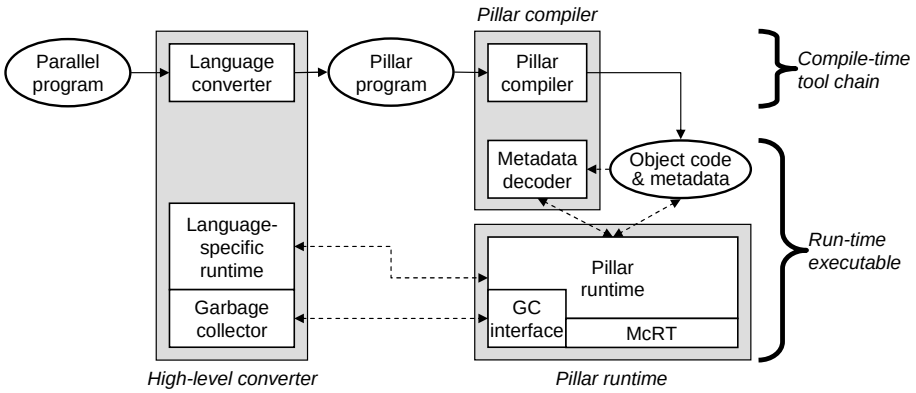


**Fig. 1.** The Pillar architecture and software stack.

To implement a parallel language using Pillar, a programmer first creates a *high-level converter* (see Fig. 1). This converter translates programs written in the parallel language into the Pillar language. Its main task is to convert constructs of the parallel language into Pillar constructs. The Pillar language is based on C and includes a set of modern sequential and parallel features (see Section 2). Since the Pillar compiler handles the tasks of code generation and traditional compiler optimizations, creating a high-level converter is significantly easier than creating a new parallel language compiler from scratch.

The second step is to create a runtime for the high-level language that provides the specialized support needed for features of the language. We call this runtime the language-specific runtime (LSR) to distinguish it from the Pillar runtime. The LSR could be written in Pillar and make use of Pillar constructs, or could be written in a language such as C traditionally used for runtime implementation. In either case, the Pillar code generated by the converter can easily call the LSR where necessary. The LSR can also make use of Pillar's runtime that, in addition to supporting the Pillar implementation, provides a set of services for high-level languages such as stack walking and garbage collection (GC) support. The Pillar runtime is layered on top of McRT, the Multi-Core RunTime [3], which provides scheduling, synchronization, and software transactional memory services.

Once the converter and LSR are written, complete executables can be formed by compiling the converted Pillar code with the Pillar compiler to produce object code,

and then linking this with the LSR, the Pillar runtime, and McRT. The Pillar compiler produces both object code and associated metadata. This metadata is used by the Pillar runtime to provide services such as stack walking and root-set enumeration, and because of it, the code is said to be *managed*. (Pillar also supports integration with non-Pillar code, such as legacy code, which is said to be *unmanaged*.) The Pillar compiler controls the metadata format, and provides its own *metadata decoder* library to interpret it to the Pillar runtime. The metadata and decoder are also linked into the executable.

The design and implementation of Pillar is still in its early phases, and currently has a few key limitations: most notably, a cache-coherent shared-memory hardware model. Another consequence is that we are not yet in a position to do meaningful performance analysis, so this paper does not present any performance results. We intend to address these issues in the future, and we also hope to increase the range of high-level languages that can target Pillar.

The following sections focus on the Pillar language, compiler, and runtime.

## 2 The Pillar language

The Pillar language has several key design principles. First, it is a compiler target language, with the goal of mapping any parallel language onto Pillar while maintaining that language's semantics. As such, Pillar cannot include features that vary across high-level languages, like object models and type-safety rules. C++, C#, and Java, for example, are too high-level to be effective target languages, as their object models and type-safety rules are not appropriate for many languages. Therefore, of necessity, Pillar is a fairly low-level language. Although most Pillar programs will be automatically generated, expert programmers must be able to directly create Pillar programs. As a result, assembly and bytecode languages are too low-level since they are difficult even for experts to use. Although inspired by `C--` [4, 5], we decided to define Pillar as *a set of extensions to C* because then we could utilize existing optimizing C compilers to get quality implementations of Pillar quickly.

Since the Pillar language is based on C, type safety properties of the source parallel language must be enforced by the high-level converter. For example, array bounds checks might be implemented in Pillar using a combination of explicit tests and conditional branches. Similarly, null-dereference checks, divide-by-zero checks, enforcing data privacy, and restricting undesired data accesses must be done at a level above the Pillar language by the high-level converter. One notable exception is that we are working on annotations to express immutability and disambiguation of memory references.

Second, Pillar must provide support for key sequential features of modern programming languages. Examples include garbage collection (specifically, the ability to identify live roots on stack frames), stack walking (e.g., for exception propagation), proper tail calls (important when compiling functional languages), second-class continuations (e.g., for exception propagation and backtracking), and the ability to make calls between managed Pillar code and unmanaged legacy code.

Third, Pillar must also support key concurrency features of parallel languages, such as parallel thread creation, transactions, data-parallel operations, and futures. Fig. 2

summarizes the syntax of the Pillar features added to the C language. These features are described in the following sections.

**Sequential constructs**

| Feature | Syntax example |
|---|---|
| Second-class continuations | `continuation k(a, b, c):`<br>`cut to k(x, y, z);` |
| Alternate control flow | foo() `also cuts to` k1, k2;<br>foo() `also unwinds to` k3, k4;<br>foo() `never returns;` |
| Tail call | `tailcall` foo(); |
| Spans | `span TAG value { ··· }` |
| Virtual stack and destructors | `VSE(k) { ···`<br>continuation k(target):<br>···<br>cut to target;<br>} |
| GC references | `ref` obj; |
| Managed/ unmanaged calls | `#pragma managed(off)`<br>#include <stdio.h><br>`#pragma managed(on)`<br>···<br>printf(···); |

**Concurrency constructs**

| Feature | Syntax example |
|---|---|
| Pcall | `pcall(aff)` foo(a, b, c); |
| Prscall | `prscall(aff)` foo(a, b, c); |
| Futures | `fcall(aff, &st)` foo(a, b, c);<br>`ftouch(&st);`<br>`fwait(&st);` |
| Trans-actions | `TRANSACTION(k) {`<br>···<br>continuation k(reason):<br>if (reason==RETRY)<br>···<br>else if (reason==ABORT)<br>···<br>} |

**Fig. 2.** Pillar syntactic elements.

### 2.1 Sequential features

*Second-class continuations:* This mechanism is used to jump back to a point in an older stack frame and discard intervening stack frames, similar to C's setjmp/longjmp mechanism. The point in the older stack frame is called a continuation, and is declared by the `continuation` keyword; the jumping operation is called a cut and allows multiple arguments to be passed to the target continuation. For any function call in Pillar, if the target function might ultimately cut to some continuation defined in the calling function rather than returning normally, then the function call must be annotated with all such continuations (these can be thought of as all alternate return points) so that the compiler can insert additional control flow edges to keep optimizations safe.

*Virtual stack elements:* A `VSE` (virtual stack element) declaration associates a clean-up task with a block of code. The "virtual stack" terminology is explained in Section 5.2. This cleanup task is executed whenever a cut attempts to jump out of the region of code associated with the VSE. This mechanism solves a problem with traditional stack cutting (such as in `C--`) where cuts do not compose well with many other operations. For example, suppose that code executing within a transaction cuts to some stack frame outside the transaction. The underlying transactional memory system would not get notified and this is sure to cause problems during subsequent execution. By using a

VSE per transaction, the transactional memory system in Pillar is notified when a cut attempts to bypass it and can run code to abort or restart the transaction. Since cuts in Pillar compose well with all the features of Pillar, we call them *composable cuts*.

*Stack walking:* The Pillar language itself has no keywords for stack walking, but the Pillar runtime provides an interface for iterating over the stack frames of a particular thread. Pillar has the `also unwinds to` annotation on a function call for providing a list of continuations that can be accessed during a stack walk. This is useful for implementing exception propagation using stack walking, as is typical in C++, Java, and C# implementations.

*Spans:* Spans are a mechanism for associating specific metadata with call sites within a syntactic region of code, which can be looked up during stack walking.

*Root-set enumeration:* Pillar adds a primitive type called `ref` that is used for declaring local variables that should be reported as roots to the garbage collector. During stack walking these roots can be enumerated. The `ref` type may also contain additional parameters that describe how the garbage collector should treat the reference: e.g., as a direct object pointer versus an interior pointer, as a weak root, or as a tagged union that conditionally contains a root. These parameters have meaning only to the garbage collector, and are not interpreted by Pillar or its runtime. If `refs` escape to unmanaged code, they must be wrapped and enumerated specially, similar to what is done in Java for JNI object handles.

*Tail calls:* The `tailcall` keyword before a function call specifies a proper tail call: the current stack frame is destroyed and replaced with the callee's new frame.

*Calls between managed and unmanaged code:* All Pillar function declarations are implicitly tagged with the *pillar* attribute. The Pillar compiler also understands a special pragma that suppresses the *pillar* attribute on function declarations; this pragma is used when including standard C header files or defining non-Pillar functions.[1] Calling conventions and other interfacing depend on the presence or absence of the *pillar* attribute in both the caller and callee, and the Pillar compiler generates calls accordingly.

Note that spans, second-class continuations, and stack walking are `C--` constructs and are described in more detail in the `C--` specification [6].

### 2.2 Concurrency features

Pillar currently provides three mechanisms for creating new logical threads: `pcall`, `prscall`, and `fcall`. Adding the `pcall` keyword in front of a call to a function with a void return type creates a new child thread, whose entry point is the target function. Execution in the original parent thread continues immediately with the statement following the `pcall`. Any synchronization or transfer of results between the two threads should use global variables or parameters passed to the `pcall` target function.

The `prscall` keyword is semantically identical to `pcall`, but implements a *parallel-ready sequential call* [7]. `Prscalls` allow programs to specify potential parallelism without incurring the overhead of spawning parallel threads if all processors are already busy. A `prscall` initially starts running the child thread as a sequential call

[1] One particularly pleasing outcome of this syntax is that managed Pillar code and unmanaged C code can coexist within the same source files.

(the parent is suspended). However, if a processor becomes free, it can start executing the parent in parallel with the child. Thus, `prscalls` are nearly as cheap as normal procedure calls, but take advantage of free processors when they become available.

The `fcall` construct can be used to parallelize programs that have certain serializable semantics. The `fcall` annotation indicates that the call may be executed concurrently with its continuation, while allowing the call to be eagerly or lazily serialized if the compiler or runtime deems it unprofitable to parallelize it. The `st` parameter to the `fcall` is a synchronization variable, called a future, that indicates the status of the call: *empty* indicates that the call has not yet been started, *busy* indicates that the call is currently being computed, and *full* indicates that the call has completed. Two forcing operations are provided for futures: `ftouch` and `fwait`. If the future is full, both return immediately; if the future is empty, both cause the call to be run sequentially in the forcing thread; if the future is busy, `fwait` blocks until the call completes while `ftouch` returns immediately. The serializability requirement holds if, for each future, its first `ftouch` or `fwait` can be safely replaced by a call to the future's target function.

Both `prscall` and `fcall` are geared toward an execution environment where there is a great deal of available fine-grain concurrency, with the expectation that the vast majority of calls can be executed sequentially within their parents' context instead of creating and destroying a separate thread.

These three keywords take an additional *affinity* parameter [8] that helps the scheduler place related threads close to each other to, e.g., improve memory locality.

Pillar provides support for transactions. A syntactic block of code is marked as a transaction, and transaction blocks may be nested. Within the transaction block, transactional memory accesses are specially annotated, and a continuation is specified as the "handler" for those situations where the underlying transactional memory system needs the program to respond to situations like a data conflict or a user retry.

The concurrency constructs described so far relate to lightweight thread-level parallelism. To support data parallelism, we intend to add Ct primitives [9] to Pillar. These primitives express a variety of nested data-parallel operations, and their semantics allow the compiler to combine and optimize multiple such operations.

## 3 Compiler/runtime architecture

The design of the Pillar language and runtime has several consequences for the Pillar compiler's code generation. In this section, we discuss some of the key interactions between the compiler-generated code and the runtime before getting into more detailed discussion of the compiler and the runtime in the following sections.

We assume that threads are scheduled cooperatively: that they periodically yield control to each other by executing yield check operations. Our experience shows that cooperative preemption offers several performance advantages over traditional preemptive scheduling in multi-core platforms [3]. The Pillar compiler is expected to generate a yield check at each method entry and backward branch, a well-known technique that ensures yielding within a bounded time. In addition to timeslice management, cooper-

ative preemption is used on a running thread to get race-free access to a target thread's stack, for operations like root-set enumeration and prscall continuation stealing.

As we explain in Section 5, our `prscall` design allows threads to run out of stack space. This requires compiled code to perform an explicit limit check in the method prolog, jumping to a special stack extension routine if there is insufficient space for this method's stack frame. This strategy for inserting such copious limit and yield checks is likely to have a noticeable performance impact; future research will focus on mitigating these costs.

Stack walking operations like span lookup, root-set enumeration, and single-frame unwinding require the compiler to generate additional metadata for each call site. One approach would be to dictate a particular metadata format that a Pillar compiler must adhere to. Another approach, which we adopted, is to let the Pillar compiler decide on its own metadata format, and to specify a runtime interface for metadata-based operations. This means that the Pillar compiler also needs to provide its own library of metadata-decoding operations to be linked into Pillar applications, and the Pillar runtime calls those routines as necessary. We favor this latter approach because it gives the compiler more flexibility in generating optimized code.

## 4 The Pillar compiler

We implemented our prototype Pillar compiler by modifying Intel's product C compiler. The compiler consists of a front-end, middle-end, and back-end, all of which we modified to support Pillar. Fig. 3 shows the number of lines of source code (LOC) changed or added, as well as the percentage of those source lines compared to those of the original code base. The front-end modifications are relatively small, limited to recognizing new Pillar lexical and syntax elements and translating them into the high-level intermediate representation (IR). In the middle- and back-end, our changes included adding new attributes and nodes to the existing IR and propagating them through compilation phases, as well as generating additional metadata required at run time. In addition, we added new internal data structures to accommodate Pillar constructs (e.g., continuations) and the necessary new analyses and phases (e.g., GC-related analysis).

| Compiler phase | Pillar changes | Percentage of compiler code | GC-related Pillar changes |
|---|---|---|---|
| Front-end | 5 Kloc | 1.2% | 5.8% |
| Middle-end | 6 Kloc | 0.5% | 1.2% |
| Back-end | 11 Kloc | 4.2% | 20.7% |
| Total | 22 Kloc | 1.3% | 12.0% |

**Fig. 3.** Compiler modification statistics.

Some Pillar constructs are implemented as simple mappings to Pillar runtime routines. These include some explicit Pillar language constructs, such as `cut to`, `pcall`,

`prscall`, and `fcall`, as well as implicit operations such as stack limit checks, stack extension, yield checks, and managed and unmanaged transitions. The compiler may partially or fully inline calls to the runtime routines to improve performance. Fig. 4 gives an example showing how the compiler deals with some of these constructs:

1. The compiler calculates the function's frame size and generates the stack limit check and extension in the prolog (line 5).
2. For cooperative scheduling, the compiler needs to generate the yield check in the prolog and at loop back-edges (line 5 & 11).
3. For Pillar concurrency constructs (`pcall`, `prscall`, and `fcall`), the compiler maps them to corresponding Pillar runtime interface functions (line 9).
4. When calling unmanaged C functions, the compiler automatically generates the transition call to the runtime routine `prtInvokeUnmanaged` (line 10).

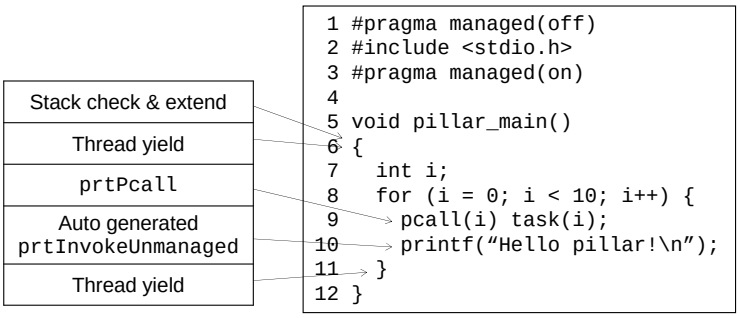


**Fig. 4.** A Pillar example.

Other Pillar constructs required deeper compiler changes. Continuation data structures must be allocated on the stack and initialized at the appropriate time. The continuation code itself must have a *continuation prolog* that fixes up the stack after a cut operation and copies continuation arguments to the right places. Continuations also affect intra-method control flow and register allocation decisions. The `VSE` and `TRANSACTION` constructs require control-flow edges into and out of the region being split so that a VSE is always pushed upon entry and popped upon exit (the push and pop operations are implemented as inlinable calls to the Pillar runtime). For every call site, the compiler must generate metadata to support runtime stack walking operations, such as unwinding a single frame, producing span data, and producing the set of live GC references. Tracking live GC references constitutes the most invasive set of changes to a C compiler, as the new `ref` type must be maintained through all IR generation and optimization phases. GC-related changes account for about 20% of the Pillar modifications in the back-end, and a very small fraction of the front-end and middle-end changes.

Some Pillar constructs need special treatment when implementing function inlining. First, the compiler cannot inline functions containing `tailcall`. Second, if the compiler decides to inline a call, and that call site contains explicit or implicit `also cuts`

`to` and `also unwinds to` annotations, then all call sites within the inlined method inherit these annotations. (Implicit `also cuts to` annotations arise from calls inside a `VSE` or `TRANSACTION` construct—there is an implicit cut edge to the destructor continuation.) Third, the compiler needs to maintain extra metadata to support intra-frame unwinding, to ensure that the stack trace looks identical regardless of inlining decisions.

Even though some deep compiler changes were required, we are pleased that the changes only amounted to about 1–2% of the code base of a highly-optimizing production C compiler, and that they preserved the compiler's traditional optimizations.[2] Of those changes, about 12% overall were related to GC, which is the single most invasive Pillar feature to implement. We believe that Pillar support could be added to other optimizing compilers at a similarly low cost.

One limitation of basing the Pillar compiler on an existing large C compiler is that we are constrained to using Pillar constructs that can be fitted onto C. It would be hard, for example, for us to support struct layout control or multiple return values. The more non-C features we choose to support, the more work we would incur in modifying the compiler to support them. We believe we have chosen a reasonable point in that spectrum for the design of Pillar.

## 5 The Pillar runtime

The Pillar runtime (PRT) provides services such as stack walking, root-set enumeration, parallel calls, stack management, and virtual stack support to compiled Pillar code and to an LSR. It is built on top of McRT [3], which the PRT relies on primarily for its scheduling and synchronization services, as well as its software transactional memory implementation [10]. The PRT interface is nearly independent of the underlying hardware: its architecture-specific properties include registers in the stack frame information returned by the stack walking support, and the machine word size. The remainder of this section provides some details on how the PRT supports its services.

### 5.1 Stack walking and root-set enumeration

The PRT provides support for walking the stack of a thread and enumerating the GC roots in its frames. To do this, PRT functions are called to (cooperatively) suspend the target thread, read the state of its topmost managed frame, then repeatedly step to the next older frame until no frames remain. At each frame, other functions can access that frame's instruction pointer, callee-saved registers, and GC roots. An additional function enumerates any roots that may be present in the thread's VSEs.

Stack walking is complicated by the need to unwind in the presence of interleaved managed and unmanaged frames. The PRT does not presume to understand the layout of unmanaged stack frames, which may vary from compiler to compiler. Instead, it uses the `VSE` mechanism to mark contiguous regions of the stack corresponding to unmanaged code, and skips over the entire region during unwinding.

[2] Note, however, that a couple of optimization phases have not yet been made Pillar-safe, and are currently disabled.

### 5.2 Composable Cuts

The PRT provides the implementation of composable cuts. These operate much like simple cuts but execute any destructor or cleanup operations of intervening VSEs.

Each thread contains a *virtual stack* of VSEs, in which the thread explicitly maintains a pointer to the virtual stack top, and each VSE contains a link to the next VSE on the stack. The continuation data structure also contains a slot for the virtual stack top. The PRT provides interfaces to push and pop VSEs. When a continuation is created, the current virtual stack top is stored in the continuation. Later, if a cut is made to this continuation, the PRT compares the current virtual stack top against the value saved in the target continuation. If these are the same, the PRT cuts directly to the target continuation. If they differ, one or more intervening frames require cleanup, and the PRT instead cuts directly to the destructor of the topmost VSE on the virtual stack, passing the original target continuation as an argument. When each VSE destructor is executed, it does its cleanup, removes itself from the virtual stack, then does another cut to the original target continuation passed to the destructor. This sequence continues until the target continuation is reached.

### 5.3 Prscalls

The Pillar compiler translates a `prscall` into a call to the PRT's prscall interface function. This function pushes a prscall VSE onto the virtual stack, copies arguments, and calls the prscall's child. Thus the child immediately starts executing sequentially. Later, an idle thread looking for work may steal the remainder of the parent's execution (unfortunately also called the parent's "continuation") by setting a continuation-stolen flag and restarting the parent. When the child terminates, it checks the continuation-stolen flag to determine whether to return to the parent or to simply exit because the continuation was stolen.

Our `prscall` design has interesting implications for stack management. When a `prscall` continuation is stolen, the stack becomes split between the parent and child threads, with the parent and child each owning one contiguous half. Since a stack can contain an arbitrary number of active `prscalls`, each of which can be stolen, a stack can become subdivided arbitrarily finely, leaving threads with tiny stacks that will quickly overflow. To deal with this, the Pillar runtime allows a thread to allocate a new "extension" stack (or "stacklet") to hold newer stack frames.

The PRT provides a stack extension wrapper that allocates a new stack (with an initial reference count of one), calls the target function, and deallocates the stack when the function returns. The stack extension wrapper also pushes a VSE whose destructor ensures that the stack will be properly deallocated in the event of a cut operation. To support stack sharing, each full stack contains a reference count word indicating how many threads are using a portion of the stack.

Logically, each `prscall` results in a new thread, regardless of whether the child runs sequentially or in parallel with its parent. When managing locks, those threads should have distinct and persistent thread identifiers, to prevent problems with the same logical thread acquiring a lock in the parent under one thread ID and releasing it under a different ID (the same is true for `pcall` and `fcall`). Each thread's persistent logical

ID is stored in thread-local storage, and locking data structures must be modified to use the logical thread ID instead of a low-level thread ID. This logical thread ID is constructed simply as the address of a part of the thread's most recent pcall or prscall VSE. As such, the IDs are unique across all logical threads, and persistent over the lifetimes of the logical threads.

### 5.4 Fcalls

The Pillar compiler translates an `fcall` into a call to the PRT's future creation function. This function creates a future consisting of a status field (empty/busy/full) and a "thunk" that contains the future's arguments and function pointer, and adds the future to the current processor's future queue. Subsequently, if a call to `ftouch` or `fwait` is made and the future's status is empty, it is immediately executed in the current thread.

At startup, the PRT creates one future pool thread for each logical processor and pins each thread to the corresponding processor. Moreover, the PRT creates a future queue for each future pool thread. A future pool thread tries to run futures from its own queue, but if the queue is empty, it will try to steal futures from other queues to balance system load.

Once the future has been evaluated, the future's thunk portion is no longer needed. To reclaim these, the PRT represents futures using a thunk structure and a separate status structure. These point to each other until the thunk is evaluated, after which the thunk memory is released. The memory for the status structure is under the control of the Pillar program, which may allocate the status structure in places such as the stack, the malloc heap, or the GC heap. We use this two-part structure so that the key part of the future structure may be automatically managed by the GC while minimizing the PRT's knowledge of the existence or implementation of the GC.

## 6 Experience using Pillar

This section describes our experience using Pillar to implement three programming languages having a range of different characteristics. These languages are Java, IBM's X10, and an implicitly-parallel functional language.

### 6.1 Compiling Java to Pillar

As part of our initial efforts, we attempted to validate the overall Pillar design through a simple Java-to-Pillar converter (JPC), leveraging our existing Java execution environment, the Open Runtime Platform (ORP) [11]. Given a trace of the Java classes and methods encountered during the execution of a program, the JPC generates Pillar code for each method from its bytecodes in the method's Java class file.

The resulting code exercises many Pillar features. First, Java variables of reference types are declared using the `ref` primitive type. Second, spans are used to map Pillar functions to Java method identifiers, primarily for the purpose of generating stack traces. They are also used, in conjunction with the `also unwinds to` annotation, to represent exception regions and handlers. Third, when an exception is thrown, ORP

uses Pillar runtime functions to walk the stack and find a suitable handler, in the form of an `also unwinds to` continuation. When the continuation is found, ORP simply invokes a `cut to` operation. Fourth, VSEs are used for synchronized methods. Java semantics require that when an exception is thrown past a synchronized method, the lock is released before the exception handler begins. A synchronized method is wrapped inside a VSE whose cleanup releases the lock. Fifth, Java threads are started via the `pcall` construct. Sixth, Pillar's managed/unmanaged transitions are used for implementing JNI calls and other calls into the ORP virtual machine.

Although several Pillar features were not exercised by the JPC, it was still effective in designing and debugging the Pillar software stack, particularly the Pillar compiler that was subjected to hundreds of thousands of lines of JPC-generated code.

### 6.2 Compiling X10 to Pillar

X10 is a new object-oriented parallel language designed for high-performance computing being developed by IBM as part of the DARPA HPCS program [12]. It is similar to Java but with changes and extensions for high-performance parallel programming. It includes asynchronous threads, multidimensional arrays, transactional memory, futures, a notion of locality (places), and distribution of large data sets.

We selected X10 because it contains a number of parallel constructs not in our other efforts, such as places, data distributions, and clocks. We also want to experiment with thread affinity, data placement, optimizing for locality, and scheduling. X10, unlike our other languages, is a good language in which to do this experimentation.

We currently compile X10 by combining IBM's open-source reference implementation of X10 [13] with the Java-to-Pillar converter. We are able to compile and execute a number of small X10 programs, and this has substantially exercised Pillar beyond that of the Java programs. In the future we will experiment with affinity and data placement.

### 6.3 Compiling a concurrent functional language

Pillar is also being used as the target of a compiler for a new experimental functional language. Functional languages perform computation in a largely side-effect-free fashion, which means that a great deal of the computational work in a program can be executed concurrently with minimal or no programmer intervention [14, 15].

Previous work has compiled functional languages to languages such as C [16], Java byte codes [17, 18], and the Microsoft Common Language Runtime (CLR) [19]. These attempts reported benefits such as interoperability, portability, and ease of compiler development. However, they have also noted the mismatches between the functional languages and the different target languages. The inability to control the object model in Java and CLR, the lack of proper tail calls, the restrictions of type safety in Java and CLR, and the inability to do precise garbage collection naturally in C, all substantially complicate compiler development and hurt performance of the final code.

`C--` and Pillar are designed to avoid these problems and provide an easy-to-target platform for functional languages. Like the Java-to-Pillar converter, our experience with the functional language showed Pillar to be an excellent target language. Pillar's lack

of a fixed object model, its support for proper tail calls, and its root-set enumeration all made implementing our functional language straightforward. Also, since Pillar is a set of C extensions, we implement most of our lowest IR directly as C preprocessor macros, and generating Pillar from this IR is straightforward. We can include C header files for standard libraries and components (e.g., the garbage collector) coded in C, and Pillar automatically interfaces the Pillar and C code. Pillar's second-class continuations are used to provide a specialized control-flow construct of the language. The stack walking-based exceptions of Java and CLR would be too heavyweight for this purpose, and C's setjmp/longjmp mechanism is difficult to use correctly and hinders performance. Implementing accurate GC in the converter is as easy as in the Java-to-Pillar converter—simply a matter of marking roots as `ref`s and using the Pillar stack walking and root-set enumeration support.

## 7 Related work

The closest language effort to Pillar is `C--` [4–6]. `C--` is intended as a low-level target language for compilers—it has often been described as a "portable assembler". Almost all Pillar features can be expressed in `C--`, but we designed Pillar to be slightly higher level than `C--`. Pillar includes, for example, `ref`s and threads instead of (as `C--` would) just the mechanisms to implement them. We also designed Pillar as extensions to C, rather than directly using `C--`, to leverage existing C compilers.

LLVM [20] is another strong and ongoing research effort whose goal is to provide a flexible compiler infrastructure with a common compiler target language. LLVM's design is focused on supporting different compiler optimizations, while Pillar is aimed at simplifying new language implementations, in part by integrating readily into an existing highly-optimizing compiler. Comparing language features, the most important differences between Pillar and LLVM are that LLVM lacks second-class continuations, spans, pcalls, prscalls, and fcalls.

C# and CLI [21, 22] are often used as intermediate languages for compiling high-level languages, and early on we considered them as the basis for Pillar. However, they lack second-class continuations, spans, prscalls, and fcalls. Furthermore, they are too restrictive in that they impose a specific object model and type-safety rules.

Pillar uses ideas from or similar to other projects seeking to exploit fine-grained parallelism without creating too many heavyweight threads. Pillar's prscalls are taken directly from Goldstein's parallel-ready sequential calls [7], which were designed to reduce the cost of creating threads yet make effective use of processors that become idle during execution. Also, like Cilk [23] and Lea's Java fork/join framework [24], Pillar uses work stealing to make the most use of available hardware resources and to balance system load. Furthermore, during prscall continuation stealing, Pillar tries to steal the earlier (deeper) continuations as Cilk does, since seizing large amounts of work tends to reduce later stealing costs. Pillar's future implementation differs from the lazy futures of Zhang et al. [25], which are implemented using a lazy-thread creation scheme similar to Pillar's prscalls. Since Pillar supports both prscalls and a separate future pool-based implementation, it will be interesting to compare the performance of both schemes for implementing futures.

## 8 Summary

We have described the design of the Pillar software infrastructure, consisting of the Pillar language, the Pillar compiler, and the Pillar runtime, as well as the high-level converter that translates programs from a high-level parallel language into Pillar. By defining the Pillar language as a small set of extensions to the C language, we were able to create an optimizing Pillar compiler by modifying only 1–2% of an existing optimizing C compiler. Pillar's thread-creation constructs, designed for a high-level converter that can find a great deal of concurrency opportunities, are optimized for sequential execution to minimize thread creation and destruction costs. Pillar's sequential constructs, many of which are taken from `C--`, have proven to be a good target for languages with modern features, such as Java and functional languages.

Our future work includes adding support for nested data parallel operations [9] to efficiently allow parallel operations over collections of data. In addition, although we currently assume a shared global address space, we plan to investigate Pillar support for distributed address spaces and message passing.

We are still in the early stages of using Pillar, but our experience to date is positive—it has simplified our implementation of high-level parallel languages, and we expect it to significantly aid experimentation with new parallel language features and implementation techniques.

## 9 Acknowledgements

We have engaged in many hours of discussions with Norman Ramsey and John Dias of Harvard University, and Simon Peyton Jones of Microsoft Research, on issues regarding Pillar and `C--`. These discussions have been instrumental in focusing the design of Pillar. We also thank the reviewers for their feedback on this paper.

## References

1. GNU: The GNU Compiler Collection See http://gcc.gnu.org/.
2. Open64: The Open Research Compiler See http://www.open64.net/.
3. Saha, B., Adl-Tabatabai, A., Ghuloum, A., Rajagopalan, M., Hudson, R., Petersen, L., Menon, V., Murphy, B., Shpeisman, T., Sprangle, E., Rohillah, A., Carmean, D., Fang, J.: Enabling Scalability and Performance in a Large Scale CMP Environment. EuroSys (March 2007)
4. Peyton Jones, S., Nordin, T., Oliva, D.: C–: A portable assembly language. In: Implementing Functional Languages 1997. (1997)
5. Peyton Jones, S., Ramsey, N.: A single intermediate language that supports multiple implementations of exceptions. Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation (June 2000)
6. Ramsey, N., Peyton Jones, S., Lindig, C.: The `C--` language specification, version 2.0. Available at http://cminusminus.org/papers.html (February 2005)
7. Goldstein, S.C., Schauser, K.E., Culler, D.E.: Lazy threads: implementing a fast parallel call. Journal of Parallel and Distributed Computing **37**(1) (1996) 5–20

8. Rajagopalan, M., Lewis, B.T., Anderson, T.A.: Thread Scheduling for Multi-Core Platforms. In: HotOS 2007: Proceedings of the Eleventh Workshop on Hot Topics in Operating Systems. (May 2007)
9. Ghuloum, A., Sprangle, E., Fang, J.: Flexible Parallel Programming for Tera-scale Architectures with Ct (2007) Available at http://www.intel.com/research/platform/terascale/TeraScale_whitepaper.pdf.
10. Saha, B., Adl-Tabatabai, A.R., Hudson, R.L., Minh, C.C., Hertzberg, B.: McRT-STM: a high performance software transactional memory system for a multi-core runtime. In: PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, New York, NY, USA, ACM Press (2006) 187–197
11. Cierniak, M., Eng, M., Glew, N., Lewis, B., Stichnoth, J.: Open Runtime Platform: A Flexible High-Performance Managed Runtime Environment. Intel Technology Journal **7**(1) (February 2003) Available at http://www.intel.com/technology/itj/archive/2003.htm.
12. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In: OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, New York, NY, USA, ACM Press (2005) 519–538
13. IBM: The Experimental Concurrent Programming Language X10. SourceForge (2007) Available at http://x10.sourceforge.net/x10home.shtml.
14. Harris, T., Marlow, S., Peyton Jones, S.: Haskell on a shared-memory multiprocessor. In: Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell, New York, NY, USA, ACM Press (2005) 49–61
15. Hicks, J., Chiou, D., Ang, B.S., Arvind: Performance studies of Id on the Monsoon Dataflow System. Journal of Parallel and Distributed Computing **18**(3) (1993) 273–300
16. Tarditi, D., Lee, P., Acharya, A.: No assembly required: compiling standard ML to C. ACM Letters on Programming Languages and Systems **1**(2) (1992) 161–177
17. Benton, N., Kennedy, A., Russell, G.: Compiling standard ML to Java bytecodes. In: ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming, New York, NY, USA, ACM Press (1998) 129–140
18. Serpette, B.P., Serrano, M.: Compiling Scheme to JVM bytecode:: a performance study. In: ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming, New York, NY, USA, ACM Press (2002) 259–270
19. Benton, N., Kennedy, A., Russo, C.V.: Adventures in interoperability: the sml.net experience. In: PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming, New York, NY, USA, ACM Press (2004) 215–226
20. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California (Mar 2004)
21. ECMA: Common Language Infrastructure. ECMA (2002) Available at http://www.ecma-international.org/publications/Standards/ecma-335.htm.
22. ISO: ISO/IEC 23270 ($C^{\#}$). ISO/IEC standard (2003)
23. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System. Journal of Parallel and Distributed Computing **37**(1) (1996) 55–69
24. Lea, D.: A Java Fork/Join Framework. In: Proceedings of the ACM 2000 Java Grande Conference, New York, NY, USA, ACM Press (2000) 36–43
25. Zhang, L., Krintz, C., Soman, S.: Efficient Support of Fine-grained Futures in Java. In: PDCS 2006: IASTED International Conference on Parallel and Distributed Computing and Systems. (November 2006)