NORTHWESTERN
UNIVERSITY

Electrical Engineering and Computer Science Department

Technical Report
NWU-EECS-09-20
December 14, 2009

Scheme with Futures: Incremental Parallelization in a Language Virtual Machine

James Swaine

Abstract

Though scripting languages have become increasingly popular in a number of computing domains, many are unable to leverage multicore architectures. This limitation exists because many virtual machine implementations for such languages were developed at a time when uniprocessor machines were ubiquitous. Thus, VM implementers saw no compelling reason to make language-level threading abstractions truly parallel. Though the absence of this requirement simplifies the virtual machine implementation, it is problematic if we wish to write parallel programs.

This thesis describes the process of modifying such a language (PLT Scheme) and its accompanying virtual machine implementation (MzScheme) to accommodate parallel constructs. MzScheme's just-in-time compiler generates blocks of machine code interspersed with call instructions targeting runtime code functions; however, blocks between these calls are mostly thread-safe. We leverage this property of JIT code to quickly modify the runtime to support multiple worker threads capable of executing thread-safe JIT code blocks; calls into unsafe runtime functions are meanwhile delegated to a single *runtime thread*.

First, we add the primitives `future` and `touch` to the language. MzScheme implements PLT Scheme's threading abstractions entirely in user space as green threads; because these threads must accommodate sophisticated control-flow constructs such as continuations, they do not map well to kernel-level threads. Thus `future` is used as the most basic means of spawning parallel tasks.

Second, we add implementation code to MzScheme to support spawning of multiple "parallel JIT code" threads. These threads perform work supplied in future invocations, executing machine code until encountering an unsafe function call. These calls are marshaled to the runtime thread, and may have the effect of serializing the remainder of the computation.

Finally, we modify core components of MzScheme individually to eliminate performance bottlenecks which cannot be avoided by simply rewriting program code. We call this process of converting a thread-safe runtime into a parallel one *incremental parallelization.*

Keywords: parallel programming, functional programming, Scheme

This project was partially supported by the National Science Foundation (NSF) via grant CNS-0709168. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any of these organizations, or the U.S. government.

Scheme with Futures: Incremental Parallelization in a Language Virtual Machine

James Swaine

December 2009

Department of Electrical Engineering and Computer Science
Northwestern University
Evanston, IL 60201

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Thesis Committee:
Peter Dinda, Northwestern University
Robby Findler, Northwestern University
Matthew Flatt, University of Utah

Keywords: parallel programming, functional programming, Scheme

Acknowledgements

I would like to thank my advisors, Peter Dinda and Robby Findler, for their immense help and support for this work. Also, many thanks to Matthew Flatt, whose knowledge of the inner workings of the MzScheme implementation, as well as his code contributions to the futures implementation itself, were invaluable. I'd also like to thank Kevin Tew, whose insight based on his own experiences with MzScheme was very helpful. His code contributions are also greatly appreciated.

Thanks to Casey Klein, who was a great source of both information on all things Scheme, and good advice in general.

I must also thank my dog, Kernel, who has kept me company through many late nights.

And finally, I am deeply grateful to my family - my wife Shasta, and my mother, father, and sister. Their support and encouragement has been endless throughout the whole process.

Contents

Listings

List of Figures

Chapter 1

Introduction

Higher-level scripting languages can greatly improve developer productivity. By automating the handling of many low-level concerns, these languages eliminate costly errors caused by invalid memory references, incorrect error code checking, and bounds checking omissions. The programmer is free to focus on program logic - *what* it does, instead of *how* it is done.

For decades, similar efforts to raise the abstraction bar have been devoted to the realm of parallel computing. Many languages have been created specifically to address this problem, but few exist which present a unified, high-level programming model for building both sequential and parallel programs. In the scripting language domain, even fewer such languages exist. In many cases it is necessary to write portions of an application using a high-level sequential language, and others in some lower-level parallel language. As we attempt to take advantage of parallelism in application logic by writing code which can use the potential of multicore processors, the multi-language mixture becomes more confused and difficult to understand.

We would like to have the ability to write parallel programs with popular scripting languages, but are often limited by their virtual machine implementations. Many scripting languages were originally implemented during a time when uniprocessor machines were ubiquitous; thus, implementers saw no compelling reason to allow multiple OS-level threads to execute simultaneously "in the runtime". Such languages typically either use a global interpreter lock or a green threads implementation to enforce this restriction.

1.1 Parallelizing Virtual Machines

Though threading libraries in this class of "single-threaded" scripting languages are useful for designing concurrent programs, they have little use if we are interested in writing parallel programs. We can propose to extend a language with new constructs for parallel programming, but must deal with the problem of adapting a pervasively sequential runtime implementation to support them (we use the terms "virtual machine" and "runtime" interchangeably throughout this thesis). Critical runtime services must be modified such that they can be safely executed concurrently by multiple threads. The sequential runtime implementation may be fraught with shared global variables to which reads and writes are not synchronized; it may be necessary to designate many portions of code as critical sections; some functions, such as garbage collection, may require all threads to halt before proceeding. In languages supporting rich control-flow structures such as continuations, implementation code may often assume that only one program stack exists - if we are to preserve the semantics of these features, the implementation must be aware that an application-level continuation might span multiple OS-level threads and hence multiple program stacks.

1.1.1 Solution 1: Retrofit Existing Code

If we are to support parallel constructs, we must address the problem of synchronized access to shared global variables. Because operating system threads within a process all share a common address space, all threads retain access to globally defined variables and may attempt to read or write them simultaneously, often leaving data corrupted or in some undefined state. One possible solution to this problem is to employ locks to synchronize access to both global variables and critical code sections, ensuring that these operations involving them are executed atomically. Though this may be a tractable approach depending on implementation code size, prior work has shown that synchronization code is notoriously difficult to both implement and debug correctly [14]. This difficulty is only exacerbated when we are attempting to retrofit existing code to be thread-safe; often the thread-safety requirement may require us to design code differently.

1.1.2 Solution 2: Start From Scratch

It may be tempting to consider rewriting the code completely, freshly designing an implementation that can support multiple threads. More programmers can be employed to accomplish this goal, because they are not required to be intimately familiar with the low-level implementation details of the original sequential version (if one exists at all).

While by no means an unrealistic option, and indeed a requirement for those who wish to build new parallel languages, this approach still suffers from a number of drawbacks. The decision to start anew with a fresh codebase can be a dangerous one, as it foregoes the reuse of any existing code in the original virtual machine implementation. Though some reuse may be possible, it can only be done in limited form - otherwise, we might have chosen solution 1. As the new codebase grows, the further it diverges from the original one, and the less opportunity there is for reuse.

Though the original code is indeed limited by its sequential nature, it is important to preserve because of the guarantees it may give us regarding *correctness*. If the virtual machine was written sufficiently long ago that its original implementers gave little or no consideration to the potential requirement of supporting multicore architectures, then we may reasonably assume that the codebase has positively evolved over many years of bug discovery by the user community. We are effectively deprecating stable, correct code in favor of new code in which bugs are sure to be endemic, a problem which is only compounded by the difficulty inherent in producing correct multithreaded code.

Languages such as Java and C# benefit from this approach - both use runtimes which have been designed "from the ground up" with concurrency in mind. But this illustrates another issue with the rewriting approach, which is also true of the retrofitting-based one: it often requires the support of a large corporation for funding, manpower, and quality assurance testing. We believe that this should not a prerequisite for successfully building parallel runtimes.

1.1.3 Solution 3: Incremental Parallelization

The first contribution of this thesis is an alternative strategy for runtime parallelization which we apply to the MzScheme machine. This approach is based on a partitioning of the set of functions provided by the

runtime into two distinct subsets: *S*, the set of functions which are already reentrant and are *safe* to execute in parallel, and *U*, the set of non-reentrant/*unsafe* functions. If we are unsure about the safety of a particular function, we conservatively assign it to *U*.

Given that this distinction can be made for all runtime functions (or rather, all functions which may be required by any user application), we can modify the implementation to allow for a restricted form of parallel execution. We use a single operating system thread, the *runtime thread*, to initialize the virtual machine as usual, and isolate all unsafe operations to that thread throughout the lifetime of the user program. Other OS-level threads are used to execute parallel code wherever indicated by the programmer, but these parallel threads depend on the runtime thread to do any unsafe work they might require.

We targeted the natural divide between JIT compiler-generated code and runtime implementation code as the boundary between safe and unsafe code in MzScheme. A typical JIT-generated machine code block in MzScheme is a sequence of instructions interspersed with `call` instructions which target C functions defined in the runtime implementation. These functions are generally sophisticated ones which cannot be inlined by the JIT compiler, and often have the most destructive side effects for internal virtual machine state. We modified the JIT compiler to eliminate generation of inlined code which directly alters or reads shared runtime state to enable the assignment of all JIT-generated code (between `call` instructions) to *S*.

With this rudimentary model for parallel execution in place, we next embarked on the "incremental" portion of the work. Achieving any degree of scalability in user programs is impossible if *U* is either large or contains functions which are likely to be frequently required by all programs, regardless of how creatively we write them. We used a process of *incremental parallelization*, in which we moved closer to a scalable implementation through a continuous cycle of writing test programs, identifying unavoidable bottleneck functions, and parallelizing them.

1.2 Parallel Scheme With Futures

The second contribution of this thesis is *Parallel Scheme with Futures*, a set of parallel extensions to the PLT Scheme programming language. Parallel Scheme is the product of our work in applying the previously

described approach in MzScheme.

1.3 Thesis Overview

The remainder of this thesis is organized as follows:

- Chapter 2 briefly reviews prior work related to parallel programming in general, with an emphasis on implementation work in scripting and functional language virtual machine environments.
- Chapter 3 presents *future* and *touch*, two fundamental language extensions added to the PLT language to enable parallel program development, and informally describes the semantics of these constructs.
- Chapter 4 explains in detail the process of modifying the MzScheme virtual machine to accommodate new parallel operators in PLT Scheme.
- Chapter 5 offers rudimentary performance measurements for several microbenchmarks written using parallel futures in PLT Scheme.
- Chapter 6 concludes the thesis and offers suggestions for future work.

1.4 Code Availability

Complete code for the latest version of the futures API is available via the PLT Scheme website:

`http://svn.plt-scheme.org/`

Chapter 2

Parallel Scheme with Futures

In this section we introduce the implementation language, PLT Scheme, as well as its underlying virtual execution environment. We describe the constructs added to the language to enable explicit parallel programming and their accompanying semantics.

2.1 The PLT Scheme Language

PLT Scheme [19] is a functional, untyped scripting language loosely based on the well-known Scheme language. The platform-independent distribution includes the MzScheme virtual machine, a graphical integrated development environment itself written in PLT Scheme [10], teaching utilities for use in introductory programming courses, and a host of other tools. The base language is accompanied by a rich set of libraries encompassing most of the functionality seen in standard libraries of other popular programming environments (JDK, C++ STL, Haskell prelude, etc.). Library functions in PLT Scheme are typically written in either PLT itself, or in less common circumstances, as C or C++ functions and exposed to PLT programs via a built-in interoperability layer. PLT Scheme also offers a powerful macro system [11] which allows users to both extend the language to encapsulate common programming idioms and create entirely new languages useful in specialized domains [21, 12, 18].

```
(future ([thunk : () -> any]) : future?

(touch ([f : future]) : any?)
```

Listing 2.1: Signatures of the *future* and *touch* operations.

```
(let* ( [x 3]
        [f1 (future (lambda () (set! x 4) x))]
        [f2 (future (lambda () (set! x 5) x))])
  (touch f1)
  (touch f2)
  (print x))
```

Listing 2.2: Parallel code with a race condition.

2.2 Parallel Extensions

We add two basic language constructs to the PLT language: *future* and *touch*. These constructs allow the development of parallel programs using a familiar fork/join programming model found in many threading libraries. The definitions of these constructs are shown in Listing 2.1.

The `future` construct accepts a parameterless λ-expression as its argument, the body of which will be executed in parallel. This function returns immediately to the caller, in a similar vein as `pthread_create` in the Pthreads API or `Delegate.BeginInvoke()` in C#. The caller receives a *future descriptor*, an opaque structure which uniquely identifies the spawned future.

The `touch` construct is roughly analogous to a join operation in the Pthreads library, or `Delegate.EndInvoke()` in C#. The calling thread supplies a future descriptor as an argument. This call will block until the future the parallel computation has completed, and will return the result value.

2.3 Synchronization

Currently, the futures API does not offer any synchronization constructs. Because the existing PLT Scheme threading module is generally orthogonal to the futures API, none of the tools offered by the threading module for designing concurrent applications will work effectively with futures. Consider the program in Listing 2.2.

```
(let* ( [x 3]
        [f1 (future (lambda () (set! x 4)))]
        [ret (touch f1)]
        [f2 (future (lambda () (set! x 5)))])
  (print (touch f2)))
```

Listing 2.3: Using *touch* as a synchronization tool.

```
(define (qsort nums)
  (let ([len (length nums)])
    (cond
      [(<= len 1) nums]
      [else
       (let* ([mid (quotient len 2)]
              [pivot (list-ref nums mid)]
              [sublists (partition nums pivot)]
              [f1 (future (lambda () (qsort (first sublists))))]
              [f2 (future (lambda () (qsort (third sublists))))])
         (append
          (touch f1)
          (second sublists)
          (touch f2)))])))
```

Listing 2.4: Parallel quicksort algorithm using futures.

The programmer has no means of controlling the order in which the variable `x` is modified. In cases such as these (though a contrived example), it is generally a good idea to decompose parallel computations into smaller ones, effectively using `touch` as a synchronization tool. For example, again in our contrived example, if we wish to force `x` to be set to `4` before `5`, we modify the code to look like Listing 2.3.

2.3.1 Nested Futures

Futures can be nested, in which parallel tasks spawn other parallel tasks in order to evenly distribute a workload across all available processors. Consider the (naive) quicksort algorithm defined in Listing 2.4, which recursively spawns parallel tasks to sort segments of the input list.

2.4 Semantics

Informally, `(future (lambda () ( ... )))` may or may not begin executing code within the supplied λ-expression in parallel on a future thread. The decision whether to execute a future in parallel may be based on a number of factors, including implementation-specific details, number of processors available, platform, and the nature of the computation. Though the futures API includes utilities to assist programmers in understanding what specific programs may be doing "under the hood", in general, it should always be assumed that a future and its spawning thread are not synchronized until an explicit `touch` call.

Chapter 3

Implementation

This section describes the implementation strategy for parallel futures in the MzScheme machine implementation, along with the challenges faced in implementing this strategy. Several features specific to PLT Scheme presented us with interesting obstacles - we enumerate them here and explain the solutions we developed to overcome them.

3.1 The MzScheme Virtual Machine

MzScheme is the C implementation of the PLT Scheme language virtual machine. Source code for the core of the runtime measures at roughly 100 KLOC, and includes a garbage collector, read-eval-print loop, just-in-time compiler, and C-level implementations for various modules which cannot otherwise be written directly in the PLT Scheme language itself. C functions in runtime implementation code may be directly exposed to the Scheme world via a built-in interoperability layer.

3.2 Separating Safe From Unsafe

In accordance with the strategy for incremental parallelization briefly outlined in the introduction, MzScheme runtime functions were roughly categorized into the *safe* and *unsafe* categories. To accomplish this, we leveraged several properties specific to the MzScheme evaluator.

PLT Scheme programs are translated in two phases. In the first phase, the original source is parsed and

```
LOAD %eax, 0x12345
...
STORE 0x12345, %eax
CALL [foobar]
LOAD %eax, 0x12345
...
```

Listing 3.1: Synchronizing global variables with registers in JIT code.

translated into a platform-independent intermediate language format. This phase can either be done dynamically at runtime, or by invoking the PLT `mzc` compiler, which translates Scheme programs into intermediate code and writes the result to a file. In the second phase, a just-in-time compiler is used to translate intermediate language code into native machine code when possible. If code requires a function which cannot be successfully inlined by the JIT compiler, a `call` to the address of the function in memory will be generated.

The MzScheme JIT compiler could be called an "aggressive" compiler. The MzScheme JIT does not dynamically decide whether or not to compile a given block or function; if the JIT is capable of inlining, it will. This behavior differs from that of other JIT compiler implementations, most notably for the JVM [20], but adds a helpful element of predictability. Generally, functions that are candidates for JIT compilation are ones that are known to be frequently used in Scheme programs and are thus critical for performance.

One goal of JIT compiler optimization is the minimization of memory references, i.e. maximal use of registers [1]. This includes minimization of references to global structures in the runtime itself. In machine code generated by the MzScheme JIT, these variables will be periodically referenced for the purpose of bidirectional "synchronization" of a register value with a shared global variable, as shown in Listing 3.1. In this example, a block of machine instructions which requires the use of a shared global variable at address `0x12345` begins by copying its contents into register `%eax`. Work then continues normally, until the code must prepare to call a function which is assumed to manipulate the global variable in some way. At this point, the contents of the `%eax` register are copied back to the location of the variable in memory, and the function is invoked. On return, we repeat the synchronization and continue.

Because global variable references in JIT code are generally confined to this type of use, the large majority of JIT-generated code could be safely assumed to only manipulate register values and user program-specific

data (stack and heap variables). Thus, it was possible to classify JIT-generated code as being thread-safe from the perspective of the virtual machine, given some way to eliminate the global state manipulation code. The solution to this problem is described in detail in Sections 3.4.2 and 3.5.1.

Hence the safe/unsafe classification arrived upon for MzScheme was to assign all functions which can be translated into machine code sequences by the JIT compiler to *S* and everything else to *U*. This allowed the safe/unsafe categories to be very cleanly separated, and eased the process of measuring the amount of unsafe work occurring in real programs. We use the terms *unsafe primitive*, *runtime function*, and *unsafe operation* interchangeably to describe non-inlinable functions in the unsafe category. When a block of JIT-generated machine code does not contain any `call` instructions to non-inlinable functions, we say it is *pure* machine code, because its execution does not have any side effects with respect to shared virtual machine state.

With a rudimentary partitioning of runtime code completed, the next task was to develop a strategy for modifying the runtime such that safe and unsafe code could be mapped to separate OS-level threads. The basic design for the parallel adaptation of MzScheme is depicted in Figure 3.1. At any point during the lifetime of an application in the MzScheme virtual machine, several kernel-level threads may be executing. We apply the following taxonomy to these threads:

- *Runtime* - the thread which was used when the MzScheme operating system process was originally spawned, e.g. the active thread when the `main` function was entered. There can only be exactly one runtime thread per MzScheme process, and all unsafe operations will be performed by this thread.
- *Worker/future* - responsible for executing parallel code in futures.

3.2.1 Blocking Operations

The model in Figure 3.1 shows several future threads running concurrently with the runtime thread, depending on it to service requests for unsafe operations.

Once a future is assigned to a kernel-level thread, it will begin executing its code until it detects an attempt to perform an unsafe operation. At this point, the future will ask the runtime thread to perform this

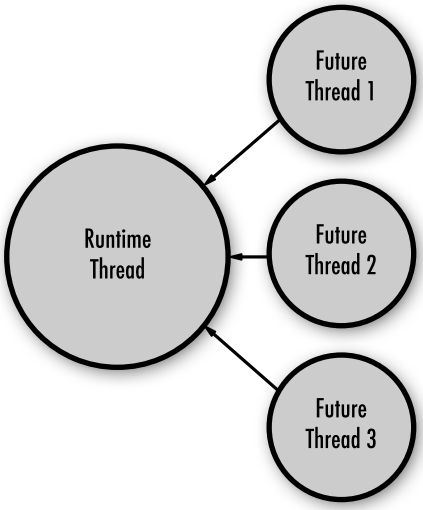

Figure 3.1: Parallel threads in MzScheme.

operation and will block until it receives an answer. Note that, because both `future` and `touch` are themselves considered unsafe primitives, these functions can only be invoked on the runtime thread.

The runtime thread does not proactively service requests for unsafe operations from future threads. After a future is spawned, no communication occurs between the spawning thread and the future thread until an explicit `touch` on the part of the user application. Upon entering a call to `touch`, if the future has not completed its computation, the calling thread will continuously poll the future for the following conditions:

- The future requires an unsafe operation in order to continue.
- The parallel computation has completed, and a result value can be returned to the `touch` caller.

If the first condition is detected, the remainder of the computation (the unsafe operation and all subsequent work) must be done sequentially with respect to the runtime thread before `touch` can return. Consider the example in Listing 3.2.

Here we define a function `add-four`, which calculates the sum of four numbers. To accomplish this we spawn a parallel future to compute one sum, while the other is computed directly on the main thread. The execution timeline for this program is shown in Figure 3.2.

```
(define (add-four a b c d)
  (let ([f1 (future (lambda ()
                      (printf "Beginning␣future␣1...~n")
                      (+ a b)))])
    (+ (+ c d) (touch f2))))
```

Listing 3.2: Sample parallel code with blocking calls.

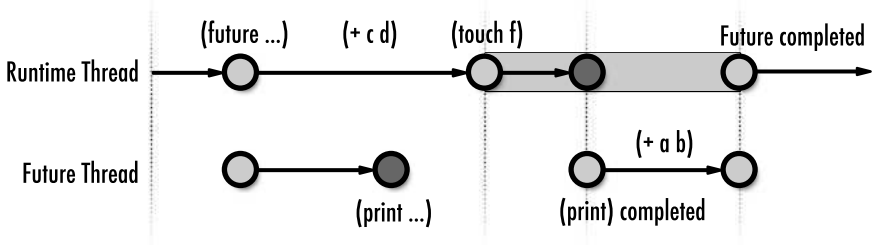

Figure 3.2: Execution timeline for single-future program with a blocking call.

Because `touch` can never be executed concurrently on multiple threads, and all requests for unsafe work are serviced within `touch` calls, no unsafe functions can ever execute concurrently. In this sense `touch` serves as a global runtime lock which is acquired on the future's first unsafe operation attempt, and released after the remainder of its computation is completed. If we were to redefine `future` as a `NOP`, and `touch` as a function which simply evaluates the future's thunk sequentially, we still retain the same ordering of unsafe primitive invocations as the original sequential implementation.

3.3 Mapping Futures to Operating System Threads

The POSIX Pthreads API was chosen for use in creating and managing future threads, for portability, one-to-one mapping of user threads to kernel threads, and the ability to use processor affinity masking to bind certain threads to certain processors.

Parallel futures are serviced by a pool of worker threads, which never grows larger than p - 1, where p is the number of processors/cores/hardware threads available in the machine. We note here that it a possible optimization may be to allow the thread pool to grow past the number of physical processors available. If it is likely that one or more future threads may be blocking waiting on an unsafe primitive request, creating

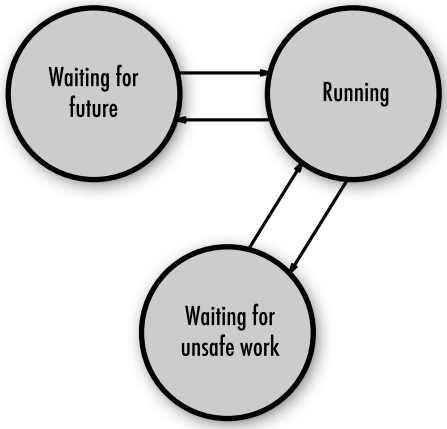

Figure 3.3: The lifetime of a parallel future thread.

additional threads and allowing them to be scheduled by the underlying operating system might improve overall performance.

When the user application spawns a future, we ask the JIT compiler to translate the supplied λ-expression into a machine code buffer and do one of the following:

- If we have not already created p - 1 threads, we create a future descriptor, add it to a *future queue*, and add a new Pthread to the worker pool. Worker threads remain live forever after creation (until the MzScheme process is terminated), continuously polling the future queue for available work. A state diagram illustrating the lifetime of a future/worker thread is given in Figure 3.3.

- If we already have p threads in the pool, we simply add the future to the queue, making it available to be serviced by any idle Pthread in the pool.

3.4 JIT Compilation

The MzScheme just-in-time compiler is primarily used to generate fast machine code to execute functions most commonly required by Scheme programs, known as *hotspots*. These include operations such as simple

```
typedef (*primitive)(int argc, Scheme_Object *argv[]);

Scheme_Object *handler(
  int argc,
  Scheme_Object *argv[],
  primitive func)
{
  Scheme_Object *retval;
  if (pthread_self() != g_runtime_thread_id)
  {
    //Block until runtime completes our work
    retval = do_runtimecall(func, argc, argv);
    return retval;
  }

  retval = func(argc, argv);
  return retval;
}
```

Listing 3.3: Typical primitive trap handler.

arithmetic, fixed-length array operations, and memory allocation (which occurs frequently in purely functional languages).

3.4.1 Unsafe Operation Detection

To enable JIT-generated machine code in parallel future threads to detect impending primitive invocations, "trap" functions were written to wrap calls to them. Function call generation code in the JIT compiler was modified to replace direct calls to primitives with calls to their respective trap functions. These functions generally took the form shown in Listing 3.3.

The handler first checks whether it is already executing on the runtime thread; if not, `do_runtimecall` is invoked, which will ask the runtime thread to complete the work for us, and block until it returns an answer. If we are already in the runtime thread, the primitive is invoked directly as usual.

Because many PLT Scheme functions have variable arity, the number of arguments a given function may accept often cannot be determined statically. C functions exposed directly to the Scheme world are generally all defined using signatures similar to the one above. This eased the process of generating trap functions for

```
[LOAD %eax, (ADDRESS IN WELL-KNOWN TL TABLE REGISTER) + OFFSET)]
...
[STORE (ADDRESS IN REGISTER) + OFFSET)]
CALL [foobar]
[LOAD %eax, (ADDRESS IN WELL-KNOWN TL TABLE REGISTER) + OFFSET)]
...
```

Listing 3.4: JIT pseudocode modified to use a thread-local table.

each different function signature we might encounter.

3.4.2 Thread-Local Tables

Figure 3.1 demonstrated the typical access pattern for a global variable reference in JIT-generated code. The presence of these types of instruction sequences in JIT-generated code renders it unsafe; race conditions or undefined behavior might ensue if code streams of this type were to be executed in parallel. We made several modifications to the JIT compiler to eliminate this problem. First, *thread-local tables* were attached to each OS-level thread. These tables are array structures where each element corresponds to a (formerly) global variable that may be referenced somewhere in JIT-generated code. In each thread, the address of the start of the array is stored in some well-known position (high up the C stack on a 32-bit machine, or in a register on a 64-bit one). Instead of of emitting instructions which refer directly to addresses of global variables, we generate code which loads the address of the thread-local table plus an offset (which corresponds to the array index of the variable we are interested in). An example of the new idiom for reading a value from a thread-local table is shown in Listing 3.4.

Because there are only 17 global variables which may be manipulated by JIT code, manually replacing code-generation logic for these variable references was relatively painless.

3.5 Garbage Collection

Because existing implementation code assumes that there can only be one active OS-level thread "in the runtime" at any one time, memory management services such as allocation and garbage collection did not

include any synchronization mechanisms. Though general operations (and their associated functions) such as memory allocation and garbage collection were easily identified as unsafe, both required special handling for correctness and performance.

3.5.1 Parallel Allocation

As an optimization for the existing sequential version of MzScheme, the JIT compiler is capable of inlining allocation code. A memory allocation request in MzScheme can take one of the following forms:

- *Fast Path* - the current page being used by the allocator has enough space to accommodate the current request. In this case, a page pointer is incremented by the size of the object being allocated, and the original page pointer is returned to the caller. This path is executed purely in machine code (with no function calls).

- *Slow Path* - the current page being used by the allocator does not have enough space to accommodate the current request. In this case, a new page must either be fetched from either the virtual machine's own internal page cache, or must be requested from the operating system. If the entire heap space has been exhausted, a garbage collection is triggered.

Modifying the standard allocation mechanism allowed us to maintain the classification of JIT-generated code blocks without function calls as being safe; it was also an excellent candidate for incremental parallelization, because in many cases a program cannot be rewritten to avoid its use. To continue allowing fast inlined allocation in parallel threads, we introduced the notion of *thread-local pages*. A future thread maintains a set of thread-local pages into which all of its small-object allocations are made (large objects are allocated into a shared heap). The formerly global page pointer was converted to a thread-local variable (a member of the thread's local table) which points to the beginning of free space in the current thread's local page. With this configuration, threads could continue to perform fast-path allocation in parallel, as shown in Figure 3.4.

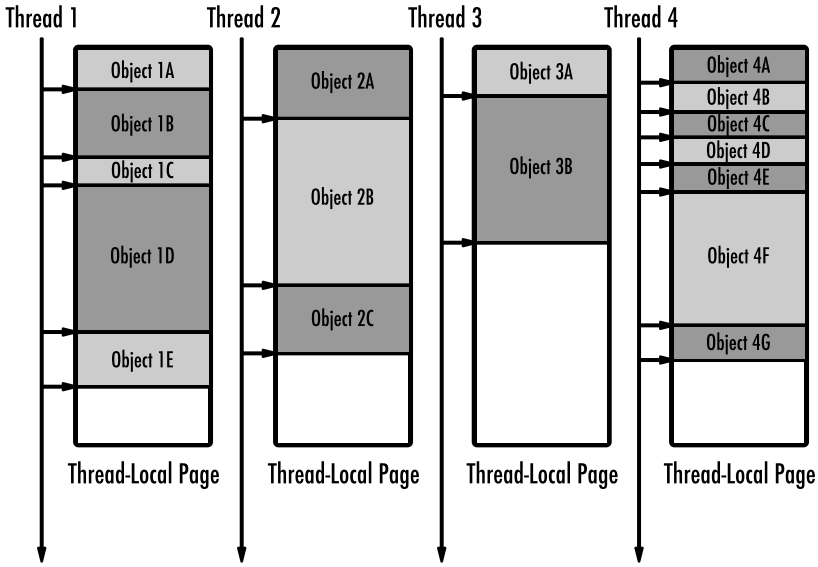

Figure 3.4: Parallel fast-path allocation with multiple threads.

Atomic Operations

In the case of a slow-path allocation, in which a thread has exhausted its existing set of pages, a new one must be atomically requested from the memory allocator. Though we treated this type of operation an unsafe one, a different mechanism was developed which allowed the runtime to proactively service thread-local page requests.

Many unsafe operations require a parallel computation to block until it is joined by the runtime thread. These types of operations treat the remainder of the parallel computation, including the unsafe operation and any others following it, as one large critical section. In some cases, however, we can forego this requirement if we know that shared data will always be in a consistent state after we perform the operation. We call these types of operations *atomic*.

As described above, PLT Scheme's user-level `thread` objects are implemented internally as cooperative threads. To ensure that starvation does not occur, the JIT compiler intersperses *fuel counter* checks throughout machine code streams. The fuel counter represents the number of "ticks" for which the current thread may

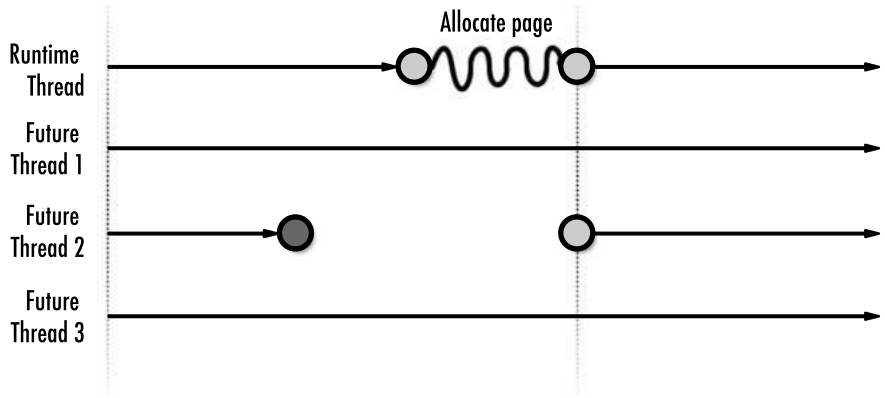

Figure 3.5: Atomic slow-path page allocation.

continue executing. If this value is greater than zero, the value will be decremented and execution will continue. Otherwise, the thread will block, yielding to the scheduler and allowing another to resume.

MzScheme's scheduler code was modified such that, each time a Scheme thread yields (or explicitly sleeps), the C function `scheme_check_future_work` is invoked. This function will execute any pending atomic operation and signal the future thread waiting on its completion that it can resume its parallel computation, as shown in Figure 3.5.

3.5.2 Rendezvous

Though the engineering overhead involved in implementing both the slow- and fast-path parallel allocation services was relatively minimal, this was not the case with the worst-case allocation scenario, in which the entire heap is exhausted and a garbage collection must occur. Ultimately, any attempt to introduce parallel garbage collection proved to fall too firmly into the "pervasive rewriting" category, which is not consistent with the philosophy behind incremental parallelization. Thus it was deemed sufficient to modify the future execution model slightly to accommodate MzScheme's stop-the-world collector.

One advantage of the slow-path parallel page allocation model is that mutation can continue to occur on parallel threads while a request is being serviced. However, this is not the case with a full garbage collection, where the collector requires that the heap remain in a constant state throughout the process. To address this problem, a *rendezvous* mechanism was developed to "stop-the-world" in a world where parallel

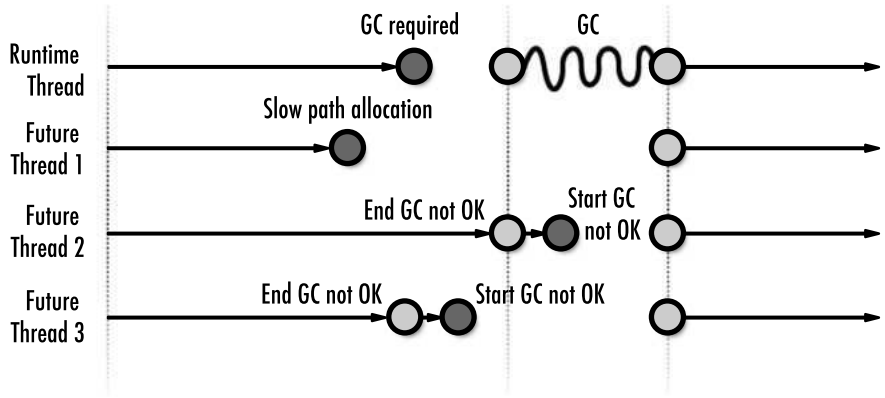

Figure 3.6: The rendezvous mechanism for stop-the-world garbage collection.

threads are present. A slow-path allocation can be further subdivided into separate paths: the usual slow path, and the "really slow path", being one that triggers a garbage collection. In the case of the really-slow-path allocation, the runtime thread, which is proactively servicing an allocation request on behalf of a parallel future, will block indefinitely, waiting for the value of the global variable `gc_ok` to equal zero. Two C functions are available to a parallel future thread, `start_gc_not_ok` and `end_gc_not_ok`, which atomically increment and decrement this counter respectively. A parallel thread must call `start_gc_not_ok` to indicate that it is preparing to execute code which may mutate the heap in some way; conversely, `end_gc_not_ok` indicates that the thread will not perform any mutation until a subsequent invocation of `start_gc_not_ok`. When `gc_ok` equals zero, all parallel threads have suspended mutation until further notice, and garbage collection may proceed. Upon beginning a garbage collection, the runtime thread raises a `wait_for_gc` flag; if a parallel thread is currently executing non-mutating code and subsequently invokes `start_gc_not_ok`, it will block waiting for `wait_for_gc` to be lowered. Thus no mutation code can run while the runtime thread is performing a collection. An example of the rendezvous mechanism is illustrated in Figure 3.6.

3.5.3 Multiple Return Values

PLT Scheme functions can accept multiple arguments and also return *multiple values*. This feature had to be addressed in the parallel futures implementation, because the existing implementation uses a special mechanism which assumes that the `scheme_current_thread` record is a global pointer to the current user-

level thread record. In modifying the JIT compiler to use thread-local tables as described in Section 3.4.2, the `scheme_current_thread` global pointer was replaced with a thread-local one.

Pure machine code sequences only reference the `current_scheme_thread` pointer for the purpose of reading and writing multiple-value results. The current Scheme thread is a convenient storage point for such values as they are communicated to and from pure machine code. In Parallel Scheme, each OS-level thread is allocated a Scheme thread "skeleton" record specifically intended for the purpose of allowing code to continue using it in this fashion.

After the user program touches a future, the future will either communicate a result value to the runtime thread, or the two threads will engage in the cyclical pattern described in Section 3.2.1, where one performs an unsafe operation, another executes machine code, and so on until the computation terminates. During this sequence, code running in either thread is capable of producing multiple return values; thus each thread must be able to communicate to the other that one is available. Because the future descriptor is the only data shared between the threads, this structure is used as a temporary storage area for these values. Each time either the runtime thread or a future thread is signaled that it may resume its work, it inspects fields in the future descriptor known to hold multiple values. If any values are found, these values are copied into the thread's `scheme_current_thread` pointer so subsequent code executed by the thread finds multiple result values where they are expected.

3.5.4 Exceptions

In Parallel Scheme, program exceptions can only be raised from within unsafe primitives - they cannot ever be raised directly from pure machine code. Exceptions are typically handled in MzScheme implementation code via C's `setjmp/longjmp`. If an operation is to be performed which may throw an exception, calling code will allocate a fresh pointer on the stack and capture a continuation via `setjmp`, storing the continuation context in the stack-allocated pointer. A field in the internal representation of the current Scheme thread, `error_buf`, is set to point to the address held by the stack-allocated pointer. If the call to `setjmp` returns zero, the operation is performed normally.

If an exception occurs while the operation is being performed, code will attempt to escape via a `longjmp` to

```
1
2   Scheme_Object *do_something(int argc, Scheme_Object *argv[])
3   {
4     Scheme_Object *retval;
5     void *tempbuf, *real_error_buf;
6     real_error_buf = scheme_current_thread->error_buf;
7     scheme_current_thread->error_buf = tempbuf;
8     if (setjmp(tempbuf))
9     {
10      /* An exception occurred, and the callee jumped here */
11      longjmp(*real_error_buf);
12    }
13    else
14    {
15      retval = suspect_func(argc, argv);
16    }
17
18    return retval;
19  }
20
21  Scheme_Object *suspect_func(int argc, Scheme_Object *argv[])
22  {
23    Scheme_Object *retval;
24
25    //...do work
26    if (exception)
27      longjmp(scheme_current_thread->error_buf);
28
29    return retval;
30  }
```

Listing 3.5: Typical exception handling code in MzScheme.

`scheme_current_thread->error_buf`, in which case the original `setjmp` call will return a non-zero value, and calling code is now aware that an exception has been raised. Code demonstrating this mechanism is shown in Listing 3.5.

Because such code is guaranteed to execute sequentially on the runtime thread, there are no safety concerns here. However, in Parallel Scheme, because another OS-level thread is waiting for a response from the runtime thread to continue, a mechanism was needed to signal that an exception has occurred during unsafe work and that the parallel computation should be cancelled.

The worker thread code used to detect exceptional conditions and cancel work prematurely is shown in

Listing 3.6. When a worker thread detects that a new future is available to be executed, it points the current Scheme thread's `error_buf` field to a temporary buffer passed to `setjmp`, as done in the usual exception-handling logic. In this case, the JIT-generated machine code block to be executed is treated as the "suspect" function which might throw an exception. If an unsafe operation is trapped, the worker thread will invoke `do_runtimecall`, as demonstrated in Section 3.4.1, to wait on the result. If an exception was thrown on the runtime thread, code there will set the future descriptor's `no_retval` field to 1 (true), which serves as the worker thread's signal to cancel the remainder of its computation.

```
1
2   /* Future/worker threads never exit this function */
3   void worker_thread_loop()
4   {
5     void *tmpbuf, *jitcode;
6     future_t *f;
7     Scheme_Object *retval;
8
9     poll_for_work:
10      if (WAIT_FOR_PENDING_FUTURE(f))
11      {
12        /* Found a pending future */
13        jitcode = jit_compile(f->thunk);
14        scheme_current_thread->error_buf = tmpbuf;
15        if (setjmp(tmpbuf))
16        {
17          /* An exception was raised on the runtime thread
18              and do_runtimecall jumped back here -
19              return a NULL value */
20          retval = NULL;
21        }
22        else
23          retval = execute(jitcode);
24
25        f->retval = retval;
26        f->status = FINISHED;
27        goto poll_for_work;
28      }
29  }
30
31  Scheme_Object *do_runtimecall(
32    Func func,
33    int argc,
34    Scheme_Object *argv[])
35  {
36    future_t *my_future = get_current_future();
37    ...Signal runtime that unsafe work is available
38    ...Wait until notified that runtime completed work
39
40    /* Unsafe work was completed */
41    if (my_future->no_retval == 1)
42    {
43      /* An exception occurred on runtime thread -
44          jump back to worker_thread_loop */
45      my_future->no_retval = 0;
46      longjmp(scheme_current_thread->error_buf);
47    }
48  }
```

Listing 3.6: Detecting exceptional conditions in a parallel future.

Chapter 4

Performance Evaluation

Several microbenchmarks were implemented using Parallel Scheme to measure both scalability and performance relative to comparable C implementations. The timing results for these microbenchmarks are given in Figures 4.1 and 4.2. All versions were compiled and executed using the following machine:

Model Name	Mac Pro
Model Identifier	MacPro3,1
Operating System	Mac OS X Snow Leopard
Processor Name	Quad-Core Intel Xeon
Processor Speed	3.2 GHz
Number of Processors	2
Total Number of Cores	8
L2 Cache (per processor)	12 MB
Memory	8 GB
Bus Speed	1.6 GHz
Boot ROM Version	MP31.006C.B05
SMC Version (system)	1.25f4

Signal Convolution

Convolution is a signal-processing algorithm used to determine the output signal of a system given its *impulse response* or *kernel*, which defines how the system will respond given an impulse function applied to the input signal. For any input signal x, we can compute each value in the corresponding output signal y using the following equation:

$$y^n = \sum_{-k}^{k} x^k \cdot h^{n-k} \tag{4.1}$$

where k is time and h is the impulse response/kernel. Our implementation computes an output signal given a one-dimensional input signal and kernel, both of which are made up of floating-point values.

We include timing measurements for two "reference" implementations of the convolution code:

- *plt* - a purely sequential version of the algorithm, executed in a vanilla MzScheme build (parallel futures disabled).
- *gcc* - a purely sequential implementation written in C and compiled with the `gcc` toolchain.

Timing results for signal convolution are displayed in the left-hand grid of Figure 4.1. The flat dotted lines in the figure indicate the running time of these two reference implementations. Convolution exhibits the best scalability of the three microkernels implemented; this can be attributed to the fact that the bulk of the work is performed in a tight (inner) loop with no allocation. The achievement of these results relied heavily on the use of unsafe floating-point operations in PLT Scheme, in which the programmer can direct the runtime to bypass expensive work such as argument verification, instead directly translating a binary operation into a corresponding machine instruction.

Mergesort

As with the image convolution microbenchmark, the mergesort algorithm was implemented in several different flavors. Two additional variants were added in this case (though as in the image convolution measurements, each of these four were only executed with one hardware thread enabled):

- *plt par* - the parallel (futures) implementation of the algorithm executed in a vanilla MzScheme build (futures disabled). PLT Scheme, by convention, is required to provide working implementations of modules regardless of specific runtime build configuration settings. Thus, a working `scheme/future` is provided in non-futures-enabled MzScheme builds. This module serves as a stub implementation, in

which `future` returns a descriptor holding the supplied thunk, and `touch` merely evaluates the future's thunk directly.

- *plt seq* - a sequential PLT Scheme implementation of a mergesort algorithm optimized for sequential execution.
- *gcc par* - a sequential C implementation of the parallel mergesort algorithm used in *plt par*.
- *gcc seq* - a sequential C implementation of the algorithm used in *plt seq*.

These additional reference measurements are intended to demonstrate the sequential overhead introduced by the parallel algorithm. Results for the mergesort implementation are displayed in the right-hand grid of Figure 4.1. The results were interesting in that the implementation generally yielded the greatest speedup on power-of-two increases in the number of hardware threads, indicating that the algorithm performed best when the number of available parallel threads allowed a subdivision of the workload into a balanced binary tree.

Sparse Matrix-Vector Multiplication

Sparce matrices are matrices in which most of the elements are zero. Because it is space-inefficient to store large numbers of elements known to be zero, sparce matrices are typically represented such that only nonzero elements occupy space in memory. We show timing results for a Parallel Scheme implementation of a sparse matrix- dense vector multiplication algorithm in Figure 4.2. In such an algorithm, we return a vector containing the dot-products of each row of the matrix and the dense vector. Our specific implementation borrows the compressed row format storage scheme, along with its associated algorithm, for the sparse matrix introduced by Blelloch et al. [4]. We briefly review this algorithm here.

A matrix is defined as a structure with the following fields:

- `num_rows` - The number of rows in the matrix.
- `num_cols` - The number of columns in the matrix.

- values - An array holding all nonzero elements in the matrix. Elements are ordered in this array such that `values` appears as a concatenation of all rows in the matrix, starting at row 0.
- `column` - An array holding corresponding column indices for each value in `value`. Thus, the column index of `values[i]` = `column[i]`.
- `rowlen` - An array holding the total number of nonzero elements in each row. The number of nonzero elements in row i = `rowlen[i]`.

Though the original NESL implementation leveraged a `gather` operation implemented in hardware, we implement this operation directly in PLT Scheme using futures. We use `gather(vect, column)`, where `vect` is the input dense vector, to build a new vector containing copies of the input vector's elements. For example, if `vect` = [1,2,3] and `column` = [0,2,1,0], then `gather(vect, column)` = [1,3,2,1]. The elements of the resulting vector can be pointwise multiplied (in parallel) with the original nonzero elements of the matrix in `values`.

Because the ordering of the resulting vector from the multiplication remains consistent with that of the original `values` input, sums for each row can be computed using the `rowlen` input array to complete the dot-product calculation.

Timing results for this implementation are given in Figure 4.2. The futures version exhibits very little scaling and actually shows a negative speedup from 7 to 8 hardware threads.

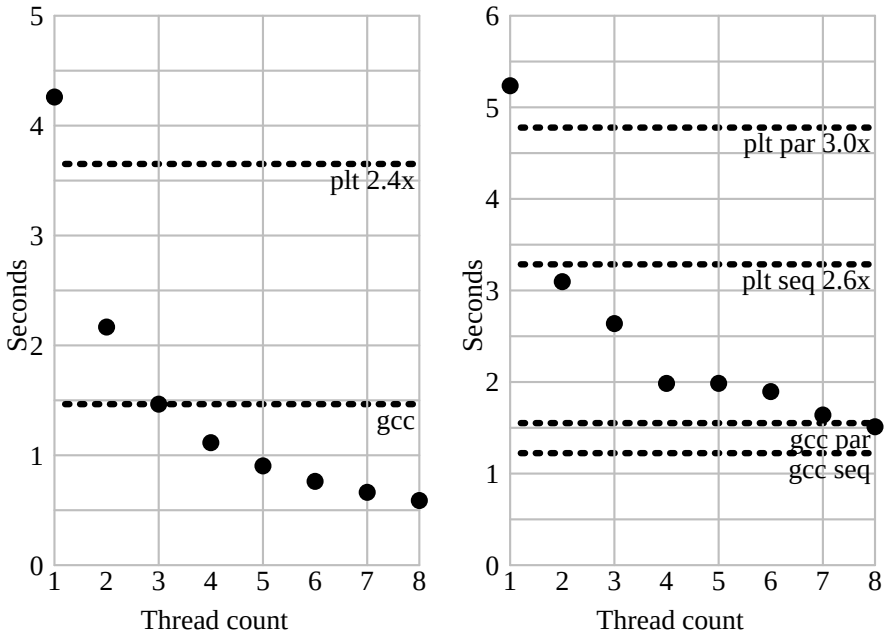

Figure 4.1: Performance results for image convolution (left) and mergesort (right).

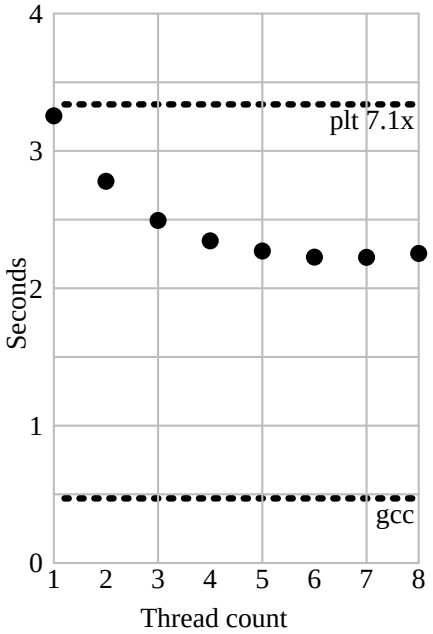

Figure 4.2: Performance results for sparse matrix-vector multiplication.

Chapter 5

Related Work

This work borrows some syntactic conventions from Multilisp [16], a parallel dialect of Scheme. Parallelism in Multilisp is also expressed primarily via the use of a `future` construct with similar usage as our own (defined in Section 2.2). However, Multilisp does not require an explicit touch on the part of the programmer - rather, touches are implicitly performed whenever an expression containing a future reference is evaluated. The Multilisp implementation described in the referenced work includes many of the elements found in more recent parallel language work, including thread-local heaps and work-stealing.

Multithreaded ERLANG [15] is a parallel adaptation of the base language's JAM runtime. Multithreaded ERLANG espouses the concept of logically isolated processing domains which may or may not map to physically separate processing elements. The system uses a hierarchical system similar to that found in X10 (described later), also borrowing from Multilisp's thread-local heaps. Garbage collections may also be performed in parallel on these thread-local heaps.

The canonical example of nested data-parallel programming is the NESL language [4]. NESL syntax might be best described as a subset of ML augmented with constructs for specifying parallel list comprehensions, as well as additional library functions which themselves perform work in parallel. The cost model of nested data-parallel languages such as NESL is given in terms of *work*, the total number of computations required by an algorithm, and *depth*, the longest chain of sequentially dependent computations in the algorithm, as opposed to machine-based models[5].

Data Parallel Haskell[7] builds on the tradition of NESL, adding nested data-parallel constructs to the

Haskell language. An interesting feature of this work is the coupling of Haskell's lazy evaluation semantics with data parallelism (where NESL uses a strict evaluation model).

The X10 language[9] is a parallel language with a syntax similar to that of Java or C/C++. X10 introduces a parallel machine abstraction based on a hierarchy of increasingly fine-grained control structures. At the top of this hierarchy is the *place*, which may be thought of as corresponding to a processing node (in a multicore machine, a single place might be mapped to a processor with multiple cores). Each place may host multiple *activities*, which are asynchronous tasks loosely analogous to threads. Activities can communicate within the same place or across places, but X10 generally restricts this communication to the sending and receiving of *value classes*, which are pass-by-value structures that can declare instance methods, similar to a `struct` in C# or a class in C++. The language includes a `future` type, which is treated as a specialization of an activity.

Fortress[2] incorporates syntax elements from ML, Java, and FORTRAN. A key concept in Fortress is *implicit parallelism.* The programmer rarely explicitly designates a certain computation as a parallel one; rather, work is expressed through "potentially parallel" constructs such as list comprehensions or map operations. For example, in the tuple assignment `(a, b, c) = (f(x), g(y), h(z))`, the functions `f`, `g`, and `h` may all be evaluated in parallel.

Chapel [8] is a high-level language built upon the philosophy that high-level parallel languages should build upon abstractions already proven to enhance developer productivity in languages such as Java and C#. Thus, support for object-oriented programming is included. Another design philosophy behind the language is the contention that compilers are not intelligent enough to always make optimal decisions regarding data placement; programmers are given some explicit control here as well. This is an interesting feature not commonly found in other high-level parallel languages other than High Performance FORTRAN, though Chapel aims to make its constructs more user-friendly.

Cilk [6] is a C-like language with parallel extensions. Interestingly, a Cilk program stripped of its Cilk-specific keywords can be compiled and executed as a garden-variety C program which preserves the same semantics as the original parallel version. Cilk uses a virtual machine with a work-stealing scheduler, which is capable of dynamically load-balancing parallel computations. The aforementioned Fortress language borrows work-stealing concepts from Cilk in evaluating implicitly parallel constructs.

The Manticore language [13] is an ML implementation augmented with both implicit and explicit parallel constructs. Implicit parallel operations are delimited with pipe characters (|). Explicit parallelism is achieved via the use of a `spawn` primitive. Manticore also allows the use of Concurrent ML constructs in the context of parallel operations. The Manticore scheduler also uses work-stealing.

Chapter 6

Conclusion and Future Work

This thesis introduced Parallel Scheme with Futures, a set of parallel extensions to the PLT Scheme programming language. This work offers two main contributions. First, a working parallel language implementation is provided, allowing the development of Scheme programs which leverage multicore/multiprocessor architectures. This implementation serves as a demonstration of the use of the incremental parallelization strategy in making a sequential runtime amenable to executing parallel languages.

We show that, though the spirit of the work lies as much in the engineering effort itself as it does with the end result, Parallel Scheme with Futures shows promise as a practical language.

6.1 Generalizing the Approach

Ultimately, incremental parallelization proved a very useful strategy; an otherwise daunting engineering task was reduced to a very tractable one. However, the question of whether incremental parallelization techniques may be applied to virtual machines other than MzScheme remains an open one. Many properties specific to the MzScheme implementation lent themselves well to the process, most notably the nearly-thread-safe nature of pure machine code; whether these properties are present in other implementations, and whether they lend themselves to the technique in the same manner as MzScheme requires further investigation. The nature of the approach makes its experimental application in other scripting language virtual machines a realistic goal. In pursuing this work with other VM implementations, we must ask the following questions:

- Is it possible to cleanly use the safe/unsafe classification scheme?

- Is implementation code structured in such a way that it would be tractable to isolate safe and unsafe code in different kernel-level threads?

6.2 Future Work

Benchmarks

The most immediate concern regarding future work is the implementation of interesting benchmarks using futures. Though the limited work done in implementing microbenchmarks is useful, it will be important to measure performance of parallel Scheme programs in the context of parallel computing in general. Several candidate benchmark suites have been identified. One possibility is the NAS Parallel Benchmarks [3], a collection of five kernels which vary in the amount of inter-thread communication required. Another candidate is the Lonestar suite of "irregular" programs [17], which provides problems which, if implemented naively, often yield very poor load balancing among available processors. Reference implementations for both suites exist in multiple languages and would provide a good basis for performance comparison.

Nested Futures

As noted in Section 3.2.1, the `future` construct is implemented with a C function at the virtual machine level and is considered an unsafe primitive. A future cannot spawn its own futures; this will have the effect of forcing the remainder of the spawning future's work to be sequential with respect to the runtime thread. This limitation is unacceptable if we are to allow nested data parallelism, in which a function applied in parallel over a set of values may themselves also execute in parallel. Thus it will be critical to treat `future` either as an atomic primitive, similar to slow-path page allocation, or even a function that is safe to invoke on any thread.

Better Abstractions

Though `future` is suitable for task-parallel programs, in general its use can sometimes be cumbersome, causing even simple programs to become unnecessarily verbose. In addition, the implementation-specific prob-

lems associated with blocking calls in futures often force the programmer to implement a given function or algorithm in an unconventional way in an effort to avoid them. In many cases, it may not be clear which portions of application code are causing the invocation of blocking runtime primitives. This can render the development of useful parallel programs intractable to the novice developer who has little knowledge of the underlying virtual machine implementation.

If parallel programming tools are to be made accessible to a wiser developer audience, better abstractions should be available. We intend futures to be used as building blocks to compose higher-level parallel programming abstractions, and have already begun to experiment in using them to add higher-level, NESL-style constructs to the PLT Scheme language. However, many questions remain — are futures a good foundation for higher levels of abstraction? If not, are we relegated to implementing them directly in runtime code itself, confining futures to a different role as a tool for task parallelism? We hope to answer these questions through exhaustive benchmark implementation, and the continued use of the incremental parallelization process wherever necessary.

Bibliography

[1] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guie-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time java compiler. *ACM SIGPLAN Notices*, pages 280–290, May 1998.

[2] E. Allen, D. Chase, J. Hallett, V. Luchangco, J. Maessen, and G. Steele. The fortress language specification. http://research.sun.com/projects/plrg/fortress.pdf.

[3] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The nas parallel benchmarks. Technical Report RNR-91-002, NASA Ames Research Center, August 1991.

[4] Guy Blelloch. Implementation of a portable nested data-parallel language. *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–111, 1993.

[5] Guy Blelloch. Programming parallel algorithms. *Communications of the ACM*, pages 85–97, March 1996.

[6] Robert Blumofe, Christopher Joerg, Bradley Kuszmaul, Charles Leiserson, Keith Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *ACM SIGPLAN Notices*, pages 207–216, 1995.

[7] Manuel T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: A status report. *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, pages 10–18, 2007.

[8] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programming and the chapel language. *International Journal of High Performance Computing Applications*, pages 291–312, August 2007.

[9] Phillippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarker. X10: An object-oriented approach to non-uniform cluster computing. *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, 2005.

[10] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. Drscheme: A programming environment for scheme. *Journal of Functional Programming*, pages 159–182, 2002.

[11] Matthew Flatt. Composable and compilable macros: You want it when? *ACM SIGPLAN Notices*, pages 72–83, September 2002.

[12] Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: Closing the book on ad hoc documentation tools. *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 109–120, 2009.

[13] Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yinggi Xiao. Manticore: A heterogeneous parallel language. *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, pages 37–44, 2007.

[14] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[15] P. Hedqvist. A parallel and multithreaded erlang implementation. Master's thesis, Uppsala University, Uppsala, Sweden, June 1998.

[16] Robert H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 501–538, October 1985.

[17] M. Kulkarni, M. Burtscher, K. Pingali, and C. Cascaval. Lonestar: A suite of parallel irregular programs. *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[18] Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. *Lecture Notes in Computer Science*, chapter A Visual Environment for Developing Context-Sensitive Term Rewriting Systems, pages 301–311. Springer Berlin / Heidelberg, 2004.

[19] Plt scheme homepage. http://www.plt-scheme.org, 2009.

[20] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a java just-in-time compiler. *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 14–18, October 2001.

[21] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 395–406, 2008.