The Design Rationale for Multi-MLton

Suresh Jagannathan, Armand Navabi, KC Sivaramakrishnan, Lukasz Ziarek
{suresh, anavabi, chandras, lziarek}@cs.purdue.edu

Abstract

Multi-MLton is a compiler and runtime environment that targets scalable multicore platforms. It combines new language abstractions and associated compiler analyses for expressing and implementing various kinds of fine-grained parallelism (safe futures, speculation, transactions, etc.), along with a sophisticated runtime system tuned to efficiently handle large numbers of lightweight threads.

Multi-MLton defines a programming model in which threads primarily communicate via message-passing. It differs from other message-passing systems insofar as the abstractions it provides permit (a) the expression of *isolation* of communication effects among groups of communicating threads; (b) composable *speculative* actions that are message-passing aware; (c) the construction of *asynchronous events* that seamlessly integrate abstract asynchronous communication protocols with abstract CML-style events, and (d) *deterministic* concurrency within threads to enable the extraction of additional parallelism when feasible and profitable.

These abstractions are supported by a combination of compile-time analyses and specialized runtime structures to enable efficient execution on scalable multicore and manycore platforms.

1. Motivation

Multi-MLton is an active ongoing project at Purdue that builds upon the MLton whole-program compiler and runtime infrastructure. Its overarching goal is to develop new formalisms and techniques to describe, optimize, and execute high-level concurrent programs, whose threads of control communicate using sophisticated message-passing protocols.

Multi-MLton 's design rationale is influenced by current trends in CMP processor, and operating systems, design. For example, Intel's recently announced SCC (Single-Chip Cloud Computer) is a manycore CPU that features 24 tiles comprised of dual-core x86 IA processors. Notably, there is *no* shared L2 cache among these tiles; instead, communication across cores is via hardware-assisted message-passing over a 2D high-bandwidth mesh network. Thus, the SCC does not support uniform access memory – application performance is dictated by the degree of affinity that exists between threads and the data they access.

At the software level, Barrelfish [2] is a new operating system kernel design that treats the underlying machine as a network of independent cores, and assumes no inter-core sharing at the lowest level. It recasts all OS services, including memory management and inter-core communication, in terms of message-passing, arguing that such a reformulation leads to improved scalability and efficiency due to additional pipelining and batching opportunities.

Well before the advent of multicore, there has been a long history of seminal research that has explored the foundations of message-passing [3, 4]. Indeed, languages like Concurrent ML [7], Erlang [1], Scala [6], and F# [9] are successful current manifestations (in varying degrees) of these models. Multi-MLton extends these efforts by integrating new forms of concurrency control into ML, and providing aggressive runtime support for true lightweight threading on an execution platform comprised of potentially hundreds of cores.

A functional programming discipline, combined with explicit communication via messages (rather than implicit communication via shared-memory), and associated lightweight concurrency primitives, offers an enticingly attractive programming model. However, there are numerous challenges to realizing this model in practice on scalable multi- and manycore platforms, with respect to both language abstractions and their implementation. It is an investigation of these challenges that guides the design of Multi-MLton .

2. Overview

We depict Multi-MLton's main features in Fig. 1.

2.1 Parasites and Lightweight Isolated Threads

At its core, Multi-MLton provides runtime support for ultra-lightweight threads called *parasites* [8]. As the name suggests, parasites live on host threads, and enable efficient sharing of thread resources. While they have potentially many uses, parasites most commonly serve as lightweight containers for synchronous communication actions. When encapsulated within a parasite, such actions become asynchronous from the perspective of the initiating computation.

In the fast path, when parasites do not block (i.e. perform synchronous communication where a partner exists, perform asynchronous communication, or purely functional computation), they only incur an overhead of a non-tail function call, and do not entail the creation of a new thread object. When a parasite does block, a new parasite on the same host can be scheduled to run; a host thread's stack representation provides appropriate metadata to the underlying scheduler to enable unblocked parasites to be executed. Compiler support facilitates reorganization of parasites within a host thread's stack, by providing bounds on its expected stack usage. Threads and parasites can be created in Multi-MLton explicitly by the application, or implicitly as a consequence of leveraging asynchronous communication primitives, described below. Regardless of the thread abstraction chosen, an application may easily initiate a large number of threads.

Preventing unwanted interactions by these threads is important, both for safety and robustness, as well as efficiency. Dynamically-specified groups of threads can be aggregated into isolated regions

that superficially resemble multi-threaded transactions. Within a given region, communication among threads is freely permitted; a thread that migrates from one region to another, however, implicitly defines a commit action that prevents it from witnessing the effects of ongoing computation in the isolated region it previously inhabited.

2.2 Asynchronous Events

Asynchrony is often used to mask communication latency by splitting the creation of a communication action from its consumption. An asynchronous action is *created* when a computation places a message on a channel (e.g., in the case of a message send), and is *consumed* when it is matched with its corresponding action (e.g., the send is paired with a receive); the creating thread may perform arbitrarily many actions before the message is consumed. In contrast, synchronous message-passing obligates the act of placing a message on a channel and the act of consuming it to take place as a single atomic action.

The challenge to building expressive asynchronous communication abstractions is defining mechanisms that allow programmers to express *composable post-creation* and *post-consumption* behavior. Even with synchronous message-passing, which conflates notions of creation and consumption, important functionality like selective communication confounds the use of simple λ-abstraction as a means of composability. This has led to the development of expressive abstractions like CML's first-class synchronous events that enable construction of composable synchronous protocols.

Supporting composable post-creation and post-consumption in an asynchronous setting introduces additional challenges because achieving such composability necessarily entails the involvement of two distinct threads of control – the thread that creates the action, and the thread that discharges it.

Multi-MLton extends CML-style synchronous events with a new family of event combinators and primitives that explicitly deal with asynchronous communication. Our extensions enable seamless composition of asynchronous protocols, and interoperate with existing CML primitives. There are two key differences between an asynchronous event primitive and its synchronous counterpart: (1) the base asynchronous primitives we define (`aSendEvt` and `aRecvEvt`) expose both creation and consumption actions – this means that internal asynchrony (e.g., threads created by a synchronous event) is never hidden within events; (2) the asynchronous counterparts to combinators like `wrap` and `guard` allow composition of post-consumption and post-creation actions of an asynchronous event.

2.3 Speculation

Speculation is one promising way to leverage large amounts of potential parallelism made available by systems comprised of hundreds of cores.

A *future* is a well-known programming construct that has typically been used to introduce concurrency to sequential programs. A computation annotated as a future is executed asynchronously and runs concurrently with its continuation. *Safe* futures [5] guarantee a future-annotated program produce the *same* (deterministic) result as its counterpart. Because of this safety guarantee, safe futures remove the burden of providing correctness in the presence of non-deterministic interleaving while affording additional parallelism. Safe futures in MultiMLton can be used to annotate any function call.

In the presence of side-effects (e.g. references, exceptions, thread creation, communication, etc.), safe futures must guarantee that any concurrent execution respects sequential semantics of the future with its continuation. Safety is ensured via a compile-time analysis and instrumentation that identifies potentially conflicting actions between a future and its continuation, along with a lightweight runtime that understands compiler-inserted instrumentation to statically enforce deterministic execution when possible.

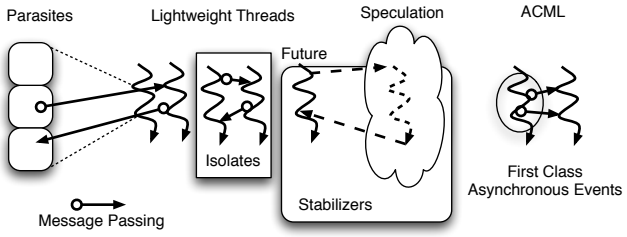

Figure 1: Abstractions found in Multi-MLton . Communication, either synchronous or asynchronous, is depicted through arrows. Parasites are shown as raw stack frames that comprise a single runtime thread.

In Multi-MLton , futures can encapsulate computation that creates new threads of control, engages in communication with other threads, etc. If a future-annotated computation violates invariants necessary to preserve determinism, its effects must be reverted. While compiler-injected barriers can sometimes be used to prevent such effects from being performed, runtime support is needed in general.

A *stabilizer* is an abstraction that can be used in conjunction with safe futures to monitor potentially unsafe speculative communication actions, and rollback their effects when necessary. Besides being used to build safe futures, stabilizers provide critical support for multithreaded software transactions in which threads comprising a transaction communicate via message-passing. The checkpoints defined by stabilizers are first-class and composable: a monitored procedure can freely create and return other monitored procedures. Stabilizers can be arbitrarily nested, and work in the presence of a dynamically-varying number of threads and non-deterministic selective communication.

3. Status

Multi-MLton has been extensively tested using the c-code generator provided by MLton on NUMA based AMD64 machines. We are working on additional support for Intel based x86-64.

References

[1] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. 2007.

[2] A. Baumann, P. Barham, P. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *SOSP*, pages 29–44, 2009.

[3] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[4] R. Milner. *Communication and Concurrency*. 1989.

[5] A. Navabi, X. Zhang, and S. Jagannathan. Dependence analysis for safe futures. In *Science of Computer Programming*, *To appear*.

[6] M. Odersky. The Scala Experiment: Can We Provide Better Language Support for Component Systems? In *POPL*, 2006.

[7] J. Reppy. *Concurrent programming in ML*. Cambridge University Press, New York, NY, USA, 1999.

[8] KC Sivaramakrishnan, L. Ziarek, R. Prasad, and S. Jagannathan. Lightweight Asynchrony using Parasitic Threads. In *DAMP*, pages 63–72, 2010.

[9] D. Syme, A. Granicz, and A. Cisternino. *Expert F#*. Apress, 2007.