Notes on the P2 Runtime

Timothy Roscoe
ETH Zürich, Switzerland

August 30, 2007

1 Introduction

This is set of notes, comments, suggestions, and proposals for the P2 runtime as a result of staring at the code in July/August 2007. This are somewhat scattered and more an aide memoire for myself than a clear report on work, but will hopefully will give some idea of where my ideas are and what I've been doing all this time.

2 Logging, tracing and debugging infrastructure

P2 currently has a somewhat disconnected story on debugging, tracing, and logging.

At one level, it would be desirable to expose the execution of P2 itself as a set of streams and/or tables of tuples which can themselves be queried, in the spirit of Atul's Eurosys paper. This is an exciting and interesting research area, but the not the focus here.

P2 still has a need for conventional debugging/tracing facility of the kind seen in most other sizeable software systems. Such a facility writes to standard out, standard error, or other output streams (such as files), and is relatively low-level. At time of writing, this functionality is provided in P2 by macros such as `TELL_WORDY`.

The basic requirements for such a facility include:

- Compilation to no code if debugging is turned off at compile time.
- Compile-time decisions made on a per-source-file basis
- Debug messages by "level" (typically an integer) and "topic" (typically an identifier for a subsystem or cross-cutting concern).
- Debugging / tracing can be turned on and off at runtime on a per-level, per-topic, or per-file basis.

- Automatic appending of file and line number information in debug messages, if required.
- Customization for trace output.
- Type flexibility: the programmer should be able to throw any object at a debug stream that she could throw at a conventional `std::ostream`.

There seems to be really no research value in this issue, it's more of an engineering issue. The open-source `libcwd` seems like a well-engineered approach to this problem. It's actively maintained, seems to address most if not all of the concerns above, and has a nice licence (Qt). Details are at `http://libcwd.sourceforge.net/`.

3 Element factories and metadata

The reflection model approach of P2 has resulted in the gradual replacement of C++ compile-time types with P2's dynamic, boxed times (`Val_Int32`, etc.) throughout the C++ code. This is most noticeable in the constructors to individual elements.

Most element class have two constructors: a "C++"-oriented constructor with a class-specific signature, and a tuple-oriented constructor which takes a single `TuplePtr` and duplicates the functionality of the first constructor, casting fields of the tuple to the correct P2 Value types, and then unboxing them to C++.

This approach has a number of deficiencies, made worse by the fact that they apply to all elements (including future ones):

- Constructor code is duplicated across the two constructors. There's no mechanism for keeping these in sync, leading to a source of insidious bugs easily missed by unit testing.
- The tuple constructor does not check types for arguments, or that the tuple is the correct size. The P2 type system will do it's casting thing, but in practice a bug-free planner should never get this wrong. We should be checking that this never happens, and flagging a bug if it does.
- While we have a catalogue representing the elements that can be created, the signature of each constructor is not captured. Thus the reflection model does not reflect (and so cannot enforce) how each element is paramterized.

This could all be fixed by having each element class's header file declare not only its name (by means of C++PP macros as present) but also its signature, and generating a factory function instead of the tuple constructor. This function actually already exists, but it calls the explicit tuple constructor.

Instead, we should generate this factory function at compile time from the signature information, and have it call the C++ constructor. This means that the accuracy of the signature declaration will be checked by the C++ compiler, and we can automatically insert runtime checks on tuple field types as well.

A mechanism like this could be extended to store more element class metadata as well, such as constraints on whether ports need to be push or pull and in which combinations.

4 The scheduler

The P2 event loop is responsible for scheduling I/O and computation within the (single-threaded) P2 process. Originally a simple select() loop as in `libasync`, it has grown to try to deal with the fixed-point semantics of Overlog and has also suffered from overzealous abstraction. It is now a bloated mess of spurious interfaces (all of which are basically standing in for closures) and special cases spanning at least 6 files. It is hard to understand. It is buggy. It is brittle.

Furthermore, the scheduler (originally the lowest-level component in the system) now depends on much higher-level functionality, including (inexplicably) the Element class.

4.1 Functionality

To step back a moment, the P2 scheduler is trying to manage execution through a sequence (or partial order) of fixed-point computations, each one of which is triggered by a (different) event tuple.

Within a fixed-point computation, dataflow operations are scheduled and executed. Much use is made of deferred procedure calls when blocking and unblocking pull and push operations. All these DPCs must execute within the fixed-point computation. The computation is "done" when there are no DPCs to execute: since no further tuples are entering the computation from outside, and no pending computations are queued within this fixed point, there is no way that the fixed-point can contain any further calculation.

The execution of each fixed point computation results in a series of actions, which are all applied after the computation completes. These actions in turn can create further (local) event tuples.

A set of external occurences (typically socket descriptor activity or timer firings) can also happen asynchronously. These should be turned into event tuples and queued as future fixed-point computations.

In the future, we may wish to execute more than one fixed-point computation in parallel (assuming that there are no data dependencies between the parallel computations). A scheduler design should not preclude evolution to support this.

A further issue is so-called "split-phase" operations, such as waiting for a disk. An open question is whether these can be handled at the level of the scheduler, or whether the problem should be left to the planner which must therefore generate the correct dataflow graphs to preserve whatever language semantics are required.

4.2 Plan

My proposed solution to this problem is a greatly simplifed event loop which, while heavily geared to the fixed-point semantic requirements above, does not depend on the rest of P2 (in particular, the dataflow model).

The state of the (single-threaded) scheduler consists of the following:

- The FDList: a set of registered file (socket) descriptors, with associated handlers for read, write, and error conditions.
- The Timer List: a list of current timer values with associated handlers, sorted by deadline.
- The Event Queue: a FIFO queue of events (handler closures). These correspond to P2 events associated with future fixed-point computations.
- A FIFO queue of Deferred Procedure Calls (DPCs). These are simply non-cancellable closures.
- A list of Actions. These are also non-cancellable closures.

The scheduler is a set of nested event loops. The outermost loop operates as follows:

1. Check for activity on any registered file descriptors. Service each active file descriptor in turn by calling its handler.
2. Check for any expired timeouts. Invoke the callback for any expired timer.
3. If the Event Queue is empty, call `select()` with a timeout equal to the next timer deadline, and then go to step 1.
4. Otherwise, remove an event from the head of the Event queue and schedule the handling of the event (see below).
5. Call `select()` with an immediate timeout (i.e. return immediately) and then go to step 1.

The event handler is a pair of inner loops. It operates as follows:

1. Call the event's closure.
2. While the DPC queue is non-empty, remove a DPC and invoke it.
3. While the action queue is non-empty, remove an action and invoke it. Action handlers are expected to themselves enqueue further events.

I believe this scheme is conceptually easy to understand, flexible, and can handle everything we need in P2 right now. It's also much simpler than what we have in place at the moment, and would obviate the need for the "runnable" concept (elements are no longer runnable, they either have a DPC queued or not).

4.3 The Flaw

This is a small problem with this to do with startup. Consider, for example, the `PullPush` element which repeatedly pulls from its one input port and pushes to its one output port. In order to start pulling, the element must have a DPC queued to notify it to start up its pull loop. This in turn will be the result of callback installed as the result of previous attempt by the element to pull, when the upstream element had indicated that it did not have a tuple ready.

This is fine except at start-of-day, when the `PullPush` element has not had a chance to install its "pull ready" callback upstream. Thus the upstream element has no function to call when it has a tuple.

This is a fundamental problem with our decision to late-bind the unblocking callbacks on pull and push, and specify them whenever pull and push are called. If these were port methods, this problem would not exist since they would be bound when the dataflow graph was created, and so would be called when the first event tuple arrived.

This latter change would require extensive changes across the code, but might be a good thing. The price paid in efficiency is that the unblock calls now have to traverse every element in a chain (as Tuples do), whereas the P2 architecture at present allows an unblock callback to leapfrog a whole sequence of elements (by passing the callback pointer unchanged through the elements).

It is not clear if this is a serious performance win, though not that this leapfrogging optimization is the default behaviour inherited from the base `Element` class (and therefore employed by any element which only overrides `simple_action()`).

One way to address this performance issue in a planner is to split the notification for a port from the port itself, and include sufficient information in the element metadata (see Section 3) to allow the plumber to bind the leapfrogging during graph instantiation.

5 Would P2 be faster with a garbage collector?

Anecdotal evidence suggests that P2 is expending considerable CPU cycles as a result of using reference-counted smart points (the `boost::Sshared_ptr` template class) to perform memory management of values and tuples (`Value` and `Tuple` objects).

The overhead comes from a number of sources. Firstly, any assignment of pointers to values or tuples involves creating a `boost::shared_ptr` object on the heap or stack, and potentially also the destruction of one of these objects.

Secondly, dereferencing any value or tuple involves two pointer indirections instead of a single indirection as before.

Thirdly, a shared pointer occupies space for the reference count (32 bits) and the "real" pointer (32 or 64 bits depending on architecture).

P2 uses shared pointers (together with the early architectural decision to make both values and tuples immutable) because they simplify greatly the decision of when to free objects. This is particularly valuable with values and tuples which are passed around between elements frequently. Alternatives to shared pointers include:

Linear types: We ensure by convention that every tuple or value assignment in the system is a "handoff": the previous "owner" of the object can no longer access it. This still means that a tuple or value has to be explicitly freed, but it can be safely freed by any process holding a pointer to it (since by definition noone else will hold such a pointer). If two entities need to both hold a particular tuple or value, it is physically copied.

Advantages of the linear types approach are that it is relatively intuitive, and is highly efficient at runtime. Downsides are that a bug (whereby a programmer tries to dereference a pointer to an object that has already been passed off to another entity) will often cause memory corruption (and core dumps).

Mark-and-sweep: We use raw pointers to values and tuples, but periodically garbage collect them. We keep track of the addresses of *all* such values we have created, and periodically mark all those that are "reachable" (in tables, held in elements, in flight, etc.). We can then free all the others at our leisure.

While a little slower than linear types at runtime (due to the mark phase), this is still highly efficient at runtime. The price to be paid is the slight pause during the marking, and extra memory occupied by the garbage values (which are no longer freed immediately) and the list of all allocated values, plus their marks.

A further downside of this approach with P2 is the need to work out the root set. With the current P2 runtime this set is non-trivial to enumerate, since tuple and value references are held be elements, tables, queues, PEL VMs, etc. all over the place. A bug in the form of an incomplete root set will result in the garbage collector freeing a value or tuple which is actually in use, with resulting corruption of main memory.

Explicit memory allocation: Both linear types and explicit mark-and-sweep are to some extent conventions on how to keep track of values and tuples - in the former case, the onus is on the programmer to make sure that a reference is never used after a handoff, whereas in the latter she must ensure that any reference to a tuple or value is reachable from the root set. Explicitly managing memory (apparently this happens in databases) therefore suffers from the same issues, and similarly requires careful and widespread code modification.

Use a garbage-collected language: It seems highly unlikely that even a really good Java VM is going to be faster than our current shared pointer implementation, but a

language which performs fast compiler-supported garbage collection like OCaml might be.

Intuitively, the cost of garbage collection over explicit memory management (whether garbage collection is performed by the language runtime or by P2-implemented mark-and-sweep scheme above) is roughly proportional to the number of tracked objects in the system over time, which itself is equal to the "steady-state" number of objects in the system plus the number of garbage objects created over a given GC epoch.

In contrast, the CPU overhead of shared pointers is proportional to the number of times that a pointer to an object is assigned (including object creation).

We actually have very little intuition about how many values and tuples a typical running P2 system creates and destroys over time, and how often such objects are passed around (i.e. how often pointers to them are assigned), and what each of these operations costs.

I decided to do a few unscientific experiments.

5.1 Numbers of values and tuples

To get a ball-park figure on how many values and tuples a modest P2 node generates, I ran the "robust ring" example from the P2 documentation with an instrumented P2 system that recorded tuple and value creation and deletion, and all assignments of shared pointers to either values or tuples (but no other shared pointer usage). I ran the canonical 3-node system for a couple of minutes, including a startup phase and a phase at the end where I killed the other two peer nodes before the one being measured. The ring is idling - there are no lookups going on here, only pings.

I tried lots of runs, but they all look roughly like Figure 1. The node has about 2000 `Value` objects and about 500 `Tuple` objects at any one time, with on average a little over 500 new `Values` being created every second (about 100 tuples), and about 1000 assignments. The periodic pinging nature of the application is clear, and so the process is somewhat bursty.

This is quite a lot for a system that's basically doing very little, but may be comparable with the number of assignment operations that a C++ program would be performing for the same periodic task.

5.2 Overhead of reference-counted pointers

I modified benchmarking program for the P2 core to measure value creation performance with and without shared pointers. The program creates a large (5000 element) array of either `boost::shared_ptr` or `Value *` and repeatedly measures how long it takes to fill the array with new values. In the former case, new values are obtained by calling `Val_`*xxx*`::mk()`; in the latter, by directly calling the constructor.

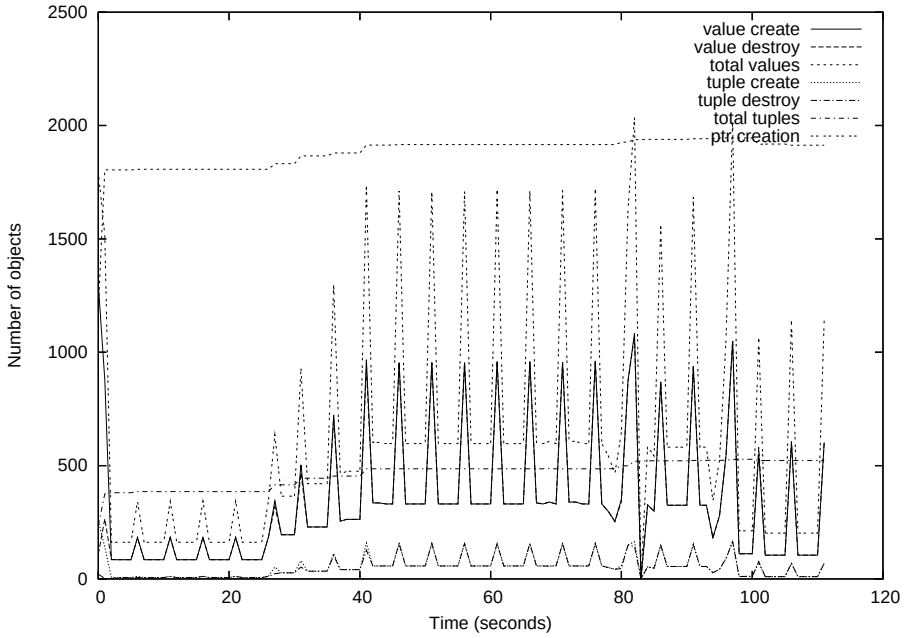

Figure 1: Value and Tuple creation, deletion, and assignment over time for the example robust ring example (one node)

The results (shown in Table 1) are fairly consistent: shared pointers are costing us about 220ns per allocation. This operation is basically an inlined assignment operator on an `ValuePtr` instance (which itself is `boost::shared_ptr<Value>`). Note that in the case of `Val_Null` we are measuring the cost of allocating a new "null" value in the raw pointer case, with the cost of creating a pointer object to the existing null singleton in the shared case, which accounts for the performance hit with raw pointers, though we note that even this is less than the cost of a new shared pointer to a non-trivial value type.

Type	shared	raw pointer	overhead
null	64	92	-28
uint64	305	86	219
double	310	88	222
string	726	367	259

Table 1: Allocation performance of P2 values with and without shared pointers (all nanoseconds). Measured on `cixous`, a dual-socket, dual-threaded (2x1x2) 3.0GHz Pentium 4 machine. The optimizer (`-O2`) is on.

5.3 Discussion

Considering only CPU cycles, the overhead of using shared pointers is a function of the rate at which pointers to tracked objects are assigned. On my hardware, using the robustRing scenario, an assignment costs about 220ns and occurs about 1000 times a second, leading to an overhead of 220μs of CPU time per second of real time.

Consider a mark-and-sweep garbage collector which collected every t seconds. The cost of the mark phase is proportional to the number of objects in the system at steady state (about 2500 in this case), plus the number created in between collections (about $500t$). We can for now assume that the cost of the sweep phase is essentially nil, since it is a loop freeing objects that would also have to be freed under any other scheme for managing memory.

Hence, a mark-and-sweep collector every t seconds results in higher CPU performance overall when:

$$220 * 10^{-6}t > (2500 + 500t) * c_m$$

i.e.:

$$c_m < \frac{220 * 10^{-6}t}{(2500 + 500t)}$$

The tradeoff (for this one specific case) is illustrated in figure 2. For example, if the cost of a mark operation on an object is 350ns, garbage collection is cheaper than reference counting if you perform a collection at most 20 seconds. Similarly, if you want a garbage collection every 5 seconds (so that no more than 2500 collectable objects are around at a time, i.e. memory usage is on average 50% higher than the reference-counted case), the cost of the mark phase amortized over each object should be about 200ns.

Obviously this is a wildly arbitrary data point. We can expect more realistic P2 applications to generate more pointer operations, and also more garbage, during a given period of time, particularly when under load. We have also not quantified the overhead introduced by smart pointer dereferencing (an extra level of indirection).

Examination of the current state of the P2 source code also suggests that determing the root set for tuples and values is non-trivial, and would entail widespread cross-cutting changes to the entire code base. This would be the case to a lesser extent if usage of tuples and values were limited to tables and passing results between elements, but P2's increasing use of reflection has meant that values and tuples are now used liberally throughout the code for initializing arbitrary other C++ objects (for example). It seems like the engineering effort of replacing the reference counts at this stage is not justifiable.

Perhaps a more useful takeaway from these initial experiments is that allowing a fast garbage-collected language (like OCaml) to manage memory would result in superior

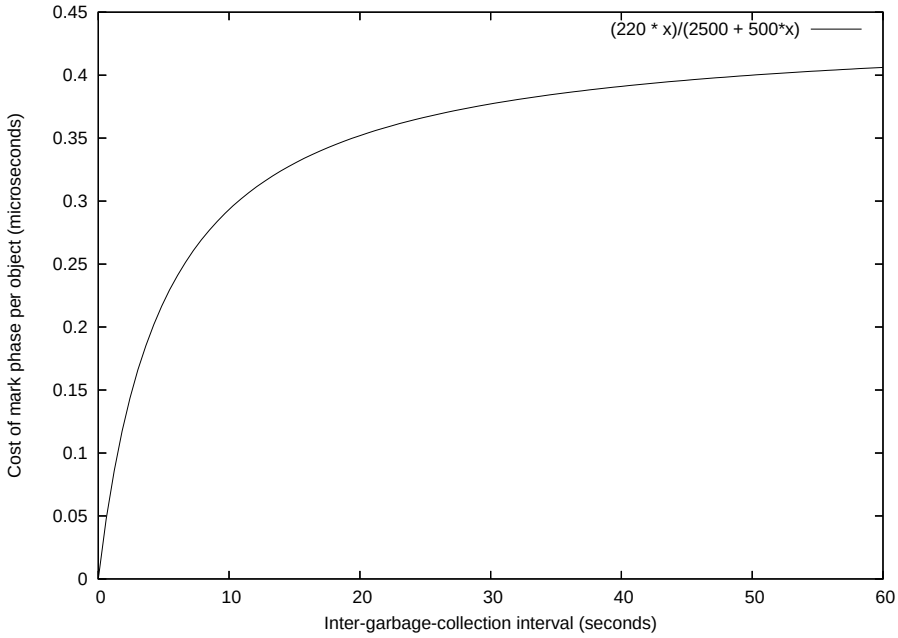

Figure 2: Tradeoff of garbage collection versus reference counting for the robust ring example.

performance to using smart pointers in C++, a conclusion in line with recent results in (for example) Melange, where compiler-directed allocation and freeing can often outperform explicit memory management.

6 P2 type system

The P2 concrete type system could do with rationalization.

- It allows 4 types of integer (32 or 64 bit, signed or unsigned), though semantically one could happily manage with signed 64-bit ints.
- While operators like plus and minus are generally polymorphic, they are not available for absolute time values and intervals - it appears that the implementor of these facilities did not attempt to fully integrate them into the rest of the type system.
- Lists, vectors, and matrices are not well handled - they are essentially opaque to the rest of the system, making it impossible to efficiently express operations on lists in Overlog (by pattern matching, for example).

7 What have I actually done?

Given the unreliability and mobility of the SVN repository for P2, I've been working out of a local Mercurial repository which I've periodically resynchronized from the Subversion using `tailor`. Aside from minor bug fixes and cleanups, there are two major changes I've made:

- I've removed all support and/or mention of `Val_Int32`, `Val_UInt32`, and `Val_UInt64` from the code. This has reduced complexity somewhat, at the cost of a slightly higher memory footprint and somewhat reduced performance on a 32-bit machine. We should also be aware that there may be some subtle lurking bugs that arise from conversions from P2 integers (which are all now signed, 64-bit quantities) to many of the integers used in the P2 code (which are generally either signed or unsigned 32-bit quantities). Caveat user.

  All the unit tests (in particular, the PEL VM tests) now pass, but with some counterintuitive results to do with how large unsigned ints are converted into signed 64-bit ints. I have put some comments in `testPel.C` which might make some of this a bit clearer.

- I've removed the scheduler, event loop, commit manager, and related files entirely. They have been replaced with `p2core/eventLoop.[Ch]`, which implements an event dispatcher similar to that described above, and `p2core/p2Net.[Ch]`, which layers above this a few asynchronous socket operations. The event loop files have extensive comments, and are also hopefully simple enough to understand by a programmer ordinarily skilled in the art. While all the code has been converted to use the new calls, it's very possible that the planner-generated dataflow doesn't quite match the model that the event loop now implements.