Integrated Performance Tuning Using Semantic Information Collected by Instrumenting the Language Runtime

Nina Engelhardt
TU Berlin
nengel@mailbox.tu-berlin.de

Sean Halle
Open Source Research Institute
Email1

Ben Juurlink
TU Berlin
b.juurlink@tu-berlin.de

Abstract

Performance tuning is an important aspect of parallel programming. After all, why go parallel, if not for higher performance? Yet existing tools leave much to be desired because they don't make the cause of performance loss evident. Only once the cause has been determined can a solution be formulated.

We describe an approach that relies upon a new model of parallel computation to link performance loss to its cause, whether that be in the hardware, runtime, or application code. The visualizations produced clearly identify idle cores, and tie the idleness to causal interactions within the runtime and hardware, and from there to the parallelism constructs that constrained the runtime and hardware behavior.

This is implemented for multi-core hardware, and we walk through a tuning session on a large multi-core machine to illustrate how performance loss is identified and how hypotheses for the cause are generated. We also give a concise description of the implementation and the computation model.

1. Introduction and Motivation

Performance visualizations and tuning tools for parallel programs are critical to achieving good performance, and yet current solutions leave something to be desired. All too often, performance tuning consists of staring at abstract views of usage or statistics, trying to *guess* the cause of performance loss. Despite many different views of performance measurements, such as frequency by line of code, message sends and receives, and core usage timelines, the user doesn't know why a function runs in a particular spot on a particular core, nor whether that is desired behavior or erroneous behavior.

Examining these myriad views can feel like being advised by the fabled six blind men examining an elephant by touching the part nearest to each**??**: one, reaching the side, says it is much like a wall; the second, feeling the tusk, compares it to a spear; the next, feeling the trunk, is reminded of a snake, and so on. All of these views are indeed correct and supply important information, but they don't connect to each other, to provide a view of how the parts interact to form a whole.

Current tools may cover all the parts of the application code, but fail to adequately connect their observations to the runtime behavior, scheduling decisions, and consequent hardware behavior. The decision about which task or virtual processor is assigned to which core at what point in time is at the heart of parallel performance, so these choices need to be highlighted and connected to the application features influencing them.

In this paper we describe a model and framework that provides a more complete picture. It captures the relevant application features (units of work and the constraints governing their execution), exposes the scheduling decisions taken in accordance with the application and other constraints, and provides a structure to which to attach the measurements captured during execution. The model relates the measurements to each other and to the application.

Using the model, we generate views that show the schedulable units in the code, runtime overhead assigned to each, the scheduling decisions made for them by the runtime, and consequent usage of the cores. Identifying idle cores is immediate, and the connections indicated by visual features enable more quickly generating the correct hypotheses for the causes of the performance losses. The visual features connect each unit to the segment of code executed, and to the constraint in the code that combined with the runtime to cause its placement in time and location. The pattern of placements, combined with contents of the code, leads to the hypothesis.

In this paper, we describe our model of computation, and illustrate its usage with a story line of performance tuning a standard parallel application on a large multi-core system.

We start with a refresher on performance tuning and an overview of previous approaches in section 2. We show usage of our visualizations through a case study in section 3, and then expand on the model behind it in section 4. Section 5 will tie the model to implementation details. Finally, we will conclude in section 6.

2. Background and Related Work

A quick review of the process of performance tuning will provide much needed context for the discussion of other tools.

Performance tuning is an iterative process that involves a mental model. The programmer takes measurements during execution that are then compared to the desired outcome. A mental model, constructed through experience and knowledge of the mechanics of execution, is used to generate a hypothesis explaining any discrepancies between the measurement and expectations. This hypothesis is then linked, again through a mental model, to things within the programmer's control, to suggest a change to make to the code. The modified code is run again, and these steps are repeated until the programmer is satisfied with the performance of the program. Thus, the mental model is central to performance tuning.

The rest of the paper will support the position that the quantities best to measure when performance tuning are scheduling decisions and the consequent usage of communication hardware and cores. Hence, the mental model should have, as concepts, units of

scheduled work, and scheduling decisions made on them, then relate those to consequent hardware behavior. The model should also relate all of those to application code, such as code boundaries that identify units, and constraints (dependencies) the application places on scheduling decisions. This model is how the individual views become integrated into a view of the whole.

With this in mind, we evaluate how well five categories of previous models used for performance tuning fit with our position, and the consequences of gaps in the fit. The first four approaches are found, sometimes in combinations, in most classic performance evaluation tools, while the fifth is starting to emerge, enabled by the growing adoption of task-based programming languages in recent years.

The commonality among the four classic approaches is models that are either hardware centric or closely related to the sequential execution model. As will be seen, the difficulty in using these approaches stems from their models, which are missing key concepts of parallel execution.

These early tools were created according to the same models that pervaded parallel application development at the time. Such models hampered the tools because they caused application code to, effectively, implement portions of the runtime as part of the application code, which hid critical information about units and constraints from the tool. Both MPI and threads cause the units of work to be *implied* by the code, which makes them difficult for tools to recognize. Likewise, constraints on scheduling are enforced by the code, but never stated in any explicit form, and so are unavailable to the tools to use to connect code features to runtime behaviors.

2.1 Thread-model based Approaches

Most of the early, more established, tools come from the threads world, and conceive of the application as a collection of virtual processors (threads) that perform actions, but don't include the concept of application-defined tasks nor constraints on them. This makes the tools unable to directly connect statistics they gather to scheduling choices and application features. That lack of connection forces the user to guess at what aspect of the code is responsible for observed performance.

Tau [] represents the thread-centric approach well. It integrates many data sources, and has rich displays. However it models cores and memories, and thread contexts, with actions taken on or by each. Because it had no well defined concept of unit of work, constraints on them, and scheduling choices, it was unable to help the user form hypotheses for the parallelism-specific causes of poor performance.

2.2 Event-centric approach

A second classic approach models parallel computation as a collection of events. Paradyn[] represents this category well. Its model of computation is based on both the timing of events and counts of events.

It has a system for user-supplied instrumentation to collect event information and it has a hypothesis mechanism that protects the user from having to write custom code to test their hypotheses. However, the hypotheses are in terms of the timing and counts of events, not the parallel computation relevant information of units of scheduled work and the scheduling decisions made on those. As such, it helps to test hypothesis but not to generate them.

2.3 Message-centric approach

Paragraph also follows an event-based model, but represents the large collection of simpler tools that instrument the MPI or other message-passing library. It shows whether cores are busy, and indicates communication overhead and flight time, but lacks an underlying parallel computation model. Without such a model, it is difficult to tie the communication pattern realized to application code features, which are what is under programmer control. Hence such message-based approaches also provide no help in forming hypotheses of which code features are responsible for observed performance.

2.4 Performance-counter approaches

Performance-counter approaches, such as VTune, POPI, and so on make up the fourth category. They concentrate on identifying hotspots and potential false-sharing, using the hardware itself as the execution model. Thus, they have no concept of unit nor scheduling event nor even runtime. As such,they do a good job of saying that something might be wrong, but don't help in pointing to what is causing the problem, and hence leave the user with no help determining what to change in their code to get better performance, with the exception of changing data structures to fix false sharing.

2.5 Newer approaches

Recent parallel languages have begun exposing units and constraints on them, for example CnC[] and StarSs[]. This provides the tools with the chance to link measurements to a model of parallel computation, such as the one we propose. The user can then link such measurements to the units defined in their code, and form hypothesis about what to change to improve performance.

As an example, the performance tuning tool[] that comes with StarSs provides a view of the code-defined units to the user. For a specific machine, it tells the user whether it thinks the task size is too small, just right, or too big. Too small has too much runtime overhead, while too big has too few tasks to keep the cores busy. This view simplifies things for the user by providing the hypothesis of what's causing poor performance and directly linking it to codde.

This is a step in the right direction, but it gives the programmer only limited information. Subtle and complex interactions between code, runtime, and hardware are often to blame for performance loss, and the limited information supplied here doesn't help with such complexities. Also, the tool is language specific, and so doesn't apply to applications in other languages.

3. Illustrative Story of Performance Tuning

In this section, we illustrate the benefits of connecting measurements to an effective computation model by walking through a typical performance tuning session. It shows the features of our approach in action, and indicates how having those features provide an advantage to the user over methods that lack them.

We first describe the program and language used, and then the features of our visualization. After this preparation, we show a sequence of the visualizations. In each, we point out how the performance loss is identified, and which visual features suggest the hypothesis for the cause of the loss. We show how this directs the user to the specific sections of code that need to be changed, and how the model helps suggest what changes to try.

3.1 The Application, and Target Hardware

In our session, we wish to tune a standard program that the reader knows well. The best example is likely matrix multiply, with which the reader should be familiar, allowing concentration on the tool without distraction about the application.

We run it on a machine with 4 sockets by 10 cores each, for a total of 40 physical cores. They are Intel WestmereEx cores running at 3.0GHz, with TurboBoost turned off.

The application code uses language features to create virtual processors (VP). The first VP created divides the work into a number of pieces and creates a new VP for each piece.

How many pieces is set by a tuning parameter in the code, and the number of cores. The application uses language features to distribute the VPs across the cores, in a round-robin fashion (before tuning).

It then creates a results VP that receives a partial-result from each piece and accumulates the results. The divider VP then waits for the results VP to indicate completion, after which the language runtime shuts down.

3.2 The language

The language used is SSR, which is based on rendez-vous style send and receive operations made between virtual processors (VPs). It has commands for creating and destroying VPs, and three kinds of send-receive paired operations.

The first, *send_from_to* specifies both sender and receiver VPs. It is used by the results VP to tell the divider VP that the work is complete. The second, *send_of_type_to*, specifies only a specific receiver, leaving the sender anonymous, which increases flexibility while maintaining some control over scope. This is used by the worker VPs doing the pieces to send their partial-result to the results processor. The third kind, *send_of_type*, only specifies the type, and so acts as a global communication channel; this is not used in our application.

The language also includes a *singleton* construct that designates a piece of code as to be executed only once, which we use to rearrange and copy data to get better cache behavior. A given copy is shared by several virtual processors on different cores, but the copy is only to be performed once.

Miscellaneous performance constructs are also available, such as one to force which core a virtual processor is assigned to. We use this in our example program to control scheduling.

A note on terminology: We often use the term "work-unit", which we define precisely, instead of "task", which has acquired multiple meanings in the literature. Work-unit is defined as the trace-segment performed on a core, between two successive scheduling events, plus the set of datums consumed by that trace segment. The word task often maps well onto this definition, and we use both words, but mean the precise work-unit definition when we say task.

3.3 The Visualizations

The first visualization is what we refer to as a scheduling consequence graph (SCG), or just consequence graph (CG). It depicts the scheduling operations performed by the runtime, and the consequent usage of the cores.

The second visualization we call the Unit & Constraint Collection, or UCC. It depicts constraints on the scheduling decisions that come from the application. Constraints can be dependencies stated explicitly in the code, or can be implied by language constructs. These limit the choices the runtime is allowed to make.

The UCC shows only application-derived information, as opposed to the consequence graph, which combines the *use* of the UCC-depicted constraints with runtime-imposed dependencies and hardware-imposed constraints. Hence, the UCC states the degrees of freedom enabled by the application, while the consequence graph states how those were made use of, by a particular runtime on particular hardware.

Fig 1 shows a consequence graph, stylized for purposes of explanation. It is composed of a number of columns, one for each core. A column represents time on the core, increasing as one goes down, measured in clock cycles. It is broken into blocks, each representing the time accounted to one work-unit. Each block is further divided into regions, each a different color, which indicates the kind of activity the core was engaged in during that region's time-span.

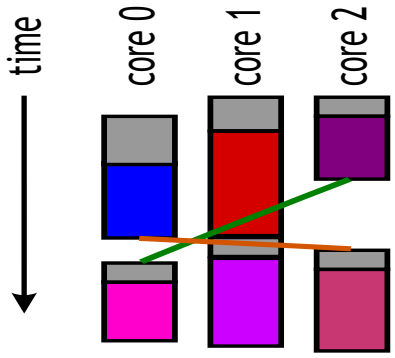


Figure 1: Stylized Scheduling Consequence Graph (SCG).

The application code executed within a block is linked to the block. In our tool, the block is labelled with a unique unitID. This ID is then linked to the code executed within that unit. In this way, the code of any block can be looked up, along with the parallelism constructs that mark the start and end of the block.

The kinds of activities within a block are defined by the computation model that underlies the visualization. The first kind of activity is the actual work, plus waiting for cache misses. It is represented by a blue-to-red region where the color indicates intensity of cache misses, with pure red representing at or above the maximum misses per instruction, and pure blue the minimum (the max and min are set in the tool that generates the visualization).

The second kind of activity is runtime overhead, represented by a gray region. This is the overhead spent on that particular work-unit. When desired by the user, it is further broken into pieces representing activities inside the runtime. The options include time spent on: constraints, when determining readiness of the work-unit; deciding which ready unit to assign to which hardware; and time spent switching from virtual processor, to the runtime, and back. In this paper, we show all runtime overhead lumped together, however in other circumstances a breakdown can be key to understanding interaction between runtime and application.

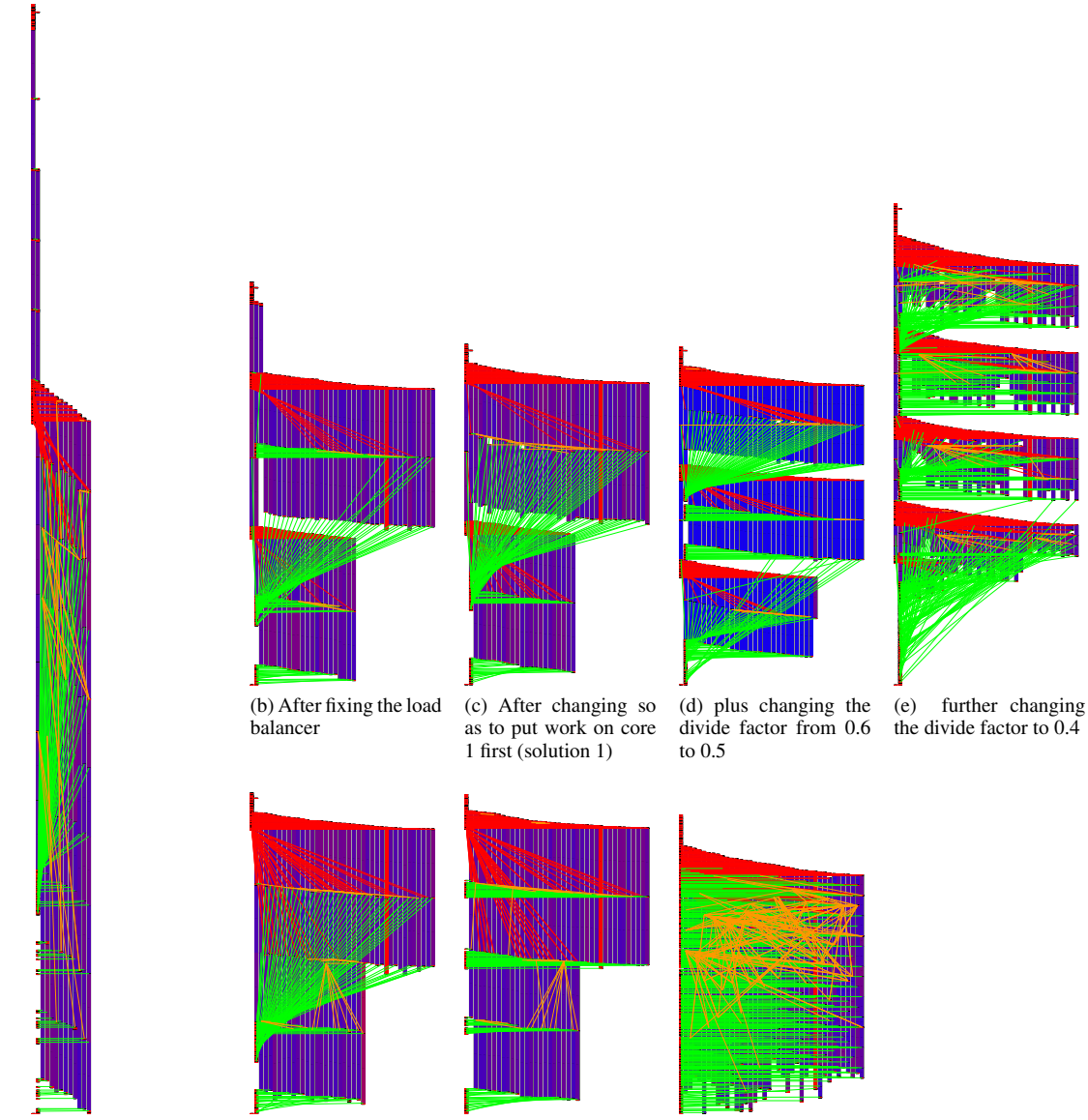
The other type of visual feature seen in Fig 1 is lines. Each represents a construct that influenced scheduling, where the color indicates which construct. A line represents two things: a constraint, whose satisfaction made the lower unit ready, and a decision by the runtime to start the lower unit on that core.

In general, lines may also be drawn that represent other kinds of interactions, which affect core usage. For example, our runtime implementation only allows one core at a time to access shared scheduling state. Visualization of this can be turned on, as additional lines linking the gray runtime regions of blocks (visualization of such interactions is turned off in this paper).

Two work-unit blocks that appear in sequence and have no lines drawn to them often have a causal dependency between them, due to the semantics of the base language. Visualization of these dependencies is also turned off, but can be inferred via the link to the code.

Note that many different orderings can be validly chosen. Which scheduler choices are valid is determined by three kinds of constraints: the application code constraints, hardware constraints, and runtime implementation imposed constraints.

The visual features allow the user to see at a glance the total execution time (height), idle cores during the run (empty columns), cache behavior (color of work regions), degree of overhead (size of gray regions), and which units constrained which other units (lines). All consequence graphs in this paper are at the same scale, so they can be compared directly.



(a) Original

(b) After fixing the load balancer

(c) After changing so as to put work on core 1 first (solution 1)

(d) plus changing the divide factor from 0.6 to 0.5

(e) further changing the divide factor to 0.4

(f) Going back to put divider VP onto its own core (Solution 2)

(g) plus moving the receive VP to same core as divider VP

(h) plus changing the divide factor to 0.4

Figure 2: Performance tuning with Scheduling Consequence Graphs (all figures drawn to the same scale)

3.4 Walk-through

In this subsection, many SCGs are shown, in Fig 2. which display the measurements collected on various runs during tuning, all on the same scale for direct comparison. They are 40 columns wide, one for each core, and relative height indicates relative execution time. They have lines in red, orange, and green, which represent application-code constructs. Red is creation of a virtual processor, green is the many-to-one `send_of_type_to`, and orange is the singleton construct. For better visibility, only constraints that cross cores are shown.

After functional debugging, the first tuning run produces the consequence graph seen in Figure 2a. The first thing to notice is that it is slimmer than expected: of the 40 available cores, only 13 are being used. Because the application places work on cores explicitly, this must be a bug in the dividing code. A cursory inspection reveals that a closing curly brace in the distribution loop had been misplaced. This may be a very simple bug, but it went unnoticed despite using this application as a test program for development of the language runtime, including performance, for several months.

3.4.1 Second Run

After fixing this, the next run (Fig 2b) corresponds much more to the expected execution behaviour. However, there remains a noticeable section at the beginning where only 3 cores have work and the other 37 remain idle.

Zooming in on those cores, we see that creation code starts running on core 0, within the creation VP, and then the next block on the core is work! Creation stops, starving the other cores. Looking at the creation code, we see that the creation VP assigns the first work VP to its own core, so that work is now waiting in the queue to execute there. When it creates the second work VP, that creation call switches core 0 to the runtime. When done with creation, the runtime takes the next VP from the queue, which is that waiting work VP. Hence core 0 does the work next instead of continuing with creation (the merits of work stealing or other scheduling strategies are independent from this illustration of how to use this approach to performance tune).

The hypothesis was generated by looking at the code linked to each block and noting the visual pattern that creation code stopped running on core 0. Work code started running instead, and only

after it finished did creation code start again. Hence, visual cues led directly to the hypothesis.

Two solutions come to mind: assign work to the other cores first, so that they would be busy when the creator VP gets interrupted, or else dedicate a core to the creator VP. The first solution has the advantage of preserving performance of the application even when run on a machine with a single-digit number of cores, so we tried it first.

3.4.2 Third run

Assigning work to the other cores first gives us Fig 2c. The section that was at the top, with idle cores, has disappeared. A small idle period can still be observed between the first and the second set of work tasks, because the work tasks have roughly the same length and the work on core 0 starts last. It thus holds up creation, which re-starts after all the others have finished work (note that work on some cores takes slightly longer because that core performs the copy-transpose singleton, and also variations are caused by cache misses).

It is also noticeable that in the second set of work units to be distributed, not enough work pieces remain to fill all cores. 16 out of 40 remain idle at the bottom.

3.4.3 Fourth and fifth runs

To try to fill the empty columns at the end, we modified the size of the work units. However, as figures 2d and 2e show, this did not help. The blank areas between "bands" of work can be seen by the red lines to be due to creation. The increased number of units causes creation to be the bottleneck again, and the time lost between sets grows larger than the time that previously was lost.

3.4.4 Sixth run

At this point we wanted to try the road not chosen, dedicating a core to the creation VP. Going back to version b of the code and implementing this solution, instead, leads to fig. 2f. The blank area between the two sets has disappeared, showing a 4% shorter execution time.

3.4.5 Seventh and eighth runs

As core 0 is now empty after the creation phase at the beginning, we also moved the receive VP there (fig. 2g). This added only a minimal improvement at this size of work unit, but allows overlapping the result collection with other work, which is an advantage when cutting the work into more pieces, requiring longer collection (fig. 2h).

Overall it is also noticeable that as work units become smaller, execution time becomes more irregular. Variability in work length correlates with the color, indicating cache behavior has worsened with smaller work size.

Note that the hypothesis, that cache behavior worsened with smaller work sizes, was generated directly from visual cues.

3.4.6 holes in the core usage

In Fig 2d, "holes" are noticeable. Inspecting these holes closer, we can see that the stalled blocks are at the end of orange lines. This indicates they are waiting upon the completion of a singleton. The pattern of blocks shows that usually the singleton unit runs before the work unit, but in these cases the singleton code was delayed until after the work on that core. This is a runtime implementation fluke. The only thing an application programmer can do is change the work size to minimize the impact. (For those curious, the first VP to reach the singleton is granted control, but a ready work VP lands in the queue during the granting activity, so when the runtime finishes granting, the work VP is next, and the VP that now owns the singleton sits and waits for the work to end. All work VPs on other cores that pass through the same singleton also wait.)

4. The Model Behind the Visualization

Now that the usage has been seen, we expand on the model behind the visualizations. The model ties the information together, and understanding it helps in generating hypotheses from the visualization features.

As seen, the model has two parts, a *Unit & Constraint Collection (UCC)*, and a *Scheduling Consequence Graph* (SCG or just consequence graph, CG). The UCC indicates the scheduling choices the application allows, and so shows what the programmer has control over. Whereas the consequence graph says which of those were actually taken during the run and the consequences of that set of choices.

We give a more precise description of UCC, then consequence graph, in turn. However, space is too limited for a complete definition, which is given in a companion paper submitted to a longer format venue.

4.1 Unit & Constraint Collection

The UCC contains all the units of work that get scheduled during a run, and all constraints the application places on scheduling those units. That's a nice solid definition, but things aren't quite that simple. The complication is that different classes of application exist, with two degrees of freedom that determine how much of the UCC is actually defined in the application vs the input data, or even in the runtime.

Some applications have everything determined in the code, with all units fixed, and all constraints fixed. An example is matrix multiply with fixed size matrices. But for others, the shape of the UCC is only partially defined by the application code. Take matrix multiply used in Section 3, where an input parameter determines the number of units created. Here, the UCC is different for each parameter value. An extreme example is an NP complete problem, with redividable units, for which the units are a function of both the input data *and* decisions made by the runtime.

We call a fully specified UCC a *concrete* UCC. Every run of an application eventually winds up defining a concrete UCC, such as seen back in Fig **??**. But the amount of UCC made concrete by the application alone falls into a two-dimensional grid. One dimension covers the units, the other the constraints.

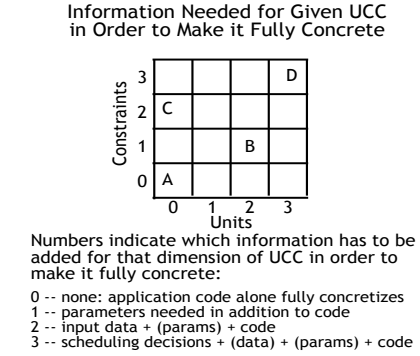


Figure 3: Abstract representation of the kinds of UCC possible. The letters A, B, C, D stand for UCCs described in the text.

Figure 3 shows the two axes and the four sets of information on each, which act as the inputs that determine the units and constraints. The position a UCC lands on the grid indicates how far it is from being fully concrete. The horizontal indicates what inputs are

still needed to determine the units, and vertical the constraints. 0 indicates that the units (constraints) are fully determined by the application code alone; 1 means parameter values also must be known; 2 means input data values also play a role, and 3 means the units (constraints) can only become known after runtime scheduling decisions have been made.

The closer the application-derived UCC is to the origin, the less additional information it needs to become concrete. The UCC labeled A in the figure is fully concrete just from the source code alone (representing for example, matrix multiply with fixed size matrices). The UCC labeled B requires the input data and parameters to be specified before its units are concrete, but just parameters to make its constraints fully concrete (as per ray-tracing, with bounce depth specified as a parameter). The UCC labeled C only has variability in its constraints, which require input data (for example, H.264 motion vectors). But even the least concrete UCC, out at the end of the diagonal (D in the figure), becomes concrete during a run of the application.

Notice, though, that a fully concrete UCC still has degrees of freedom, in which units to which hardware and the order of execution. The decisions fix interactions within the hardware, to yield the communication patterns and consequent performance seen during the run.

An added twist is that an application has a life-line, spanning from code all the way through the run, and its representation may change at the different stages of life. It starts as pristine source, then moves into specialization where code is translated into different representations than the original, and finally the specialized code is run. The UCC often changes between these points in the life-line.

For example, specialization may perform a static scheduling, which fixes the units, moving the UCC towards the origin. Alternatively, the toolchain may inject manipulator code for the runtime to use, which lets it divide units during the run when it needs more. The injection of manipulator code makes the UCC less concrete, moving it further from the origin.

The UCC still indicates what is inside the application's control vs under the runtime's control, even for applications that land far out on the diagonal. It thus indicates what can be done statically: the further out on the diagonal a UCC is, the less scheduling can be done statically in the toolchain.

In this paper, we do not suggest how to represent UCCs far out on the diagonal. One of those actually indicates a multi-verse of concrete-UCCs. Which of them materializes depends on the data that shows up and what the scheduler does. We only represent the concrete UCC that materializes during a run and leave the question of representing less concrete ones to future work.

4.2 Scheduling Consequence Graph

Whereas the UCC concentrates on application-derived information, the second part of the model adds-in effects of runtime details and hardware. It's called the Scheduling Consequence Graph because it links scheduling decisions to their performance consequences. But it also indicates the role, in the decision, of application, runtime and hardware details. As a result it identifies instances of lost performance, and links them to the cause of the loss, as seen in Section 3.

To distinguish from the UCC, the consequence graph shows the behavior resulting from scheduling decisions actually *made*, from among those *possible.* The UCC shows just the possibilities. Hence, a consequence graph shows *one* of the possible choice-sets allowed by the UCC.

A consequence graph accounts for each bit of core time. It has boxes and arcs, with the boxes divided into regions. The boxes each represent all core time assigned to one work unit, with each region inside representing a segment of time that the core was engaged in a specific type of activity. An arc links regions (or boxes) and represents a causality of some kind.

There is one kind of region for each reason that the core is being used (or being forced idle), and several kinds of arcs, one for each type of causality between regions.

The core activities associated with region types are: application work, waiting for communication of work data, managing constraints, choosing assignment of work onto cores, and runtime internals. The runtime internals have sub-categories but space is limited so we skip those here.

The arc types, representing the type of causal relationship, are: control dependency in the base language, parallel constraint that had to be satisfied (IE, one unit did something to satisfy a constraint on the other, causing it to be free to be scheduled), runtime internal causality such as a global lock (runtime on one core releases the lock, causing the other to acquire it), and arcs that represent hardware causal relationships (one work-unit finishing on a core causes another work-unit to start there, given the choice by the runtime). The formal details are given in the longer format companion paper.

We will now look at each source of causal relationship.

Constraint causality There is a constraint causality when two units are involved in a constraint, where action by one unit causes (or contributes to) satisfaction of the constraint blocking the other unit. This includes control dependencies from the base language.

Control dependencies may add superfluous constraints that eliminate some otherwise allowed choices in the UCC. An example would be a `for` loop that creates work-units – no parallelism constructs cause the creations to be done in sequence, but the base C language sequentializes it nonetheless.

Runtime internal causality Runtime implementation details may introduce "extra" causalities between units. For example, the version of VMS we instrumented for this paper runs separately on each core and relies upon a global lock for accessing shared runtime information. This lock introduces a causal relationship when the runtime on one core is attempting to process one unit, but must wait for the runtime on a different core to finish with its unit.

Normally, these are not displayed explicitly, due to clutter, but can be turned on when needed, say, to determine the cause of a particular pattern of core usage.

Hardware causality The physical fact that a given resource can only be used by one work-unit at a time introduces hardware causalities. When multiple units are free to execute, but all cores are busy, then completion of a unit on one core causes (in part) the next ready unit to run on that core.

These are also not normally displayed, due to clutter, and not all hardware dependencies are directly measured. Future work will focus on using the performance counters and other instrumentation to add more information about communication paths taken as a consequence of the scheduling decisions made. It will start with the current linkage of application-code to runtime decisions, and add consequent usage of communication hardware. This gives an end-to-end linkage between runtime choices and caused behavior on the hardware.

Consequence graph features each tie back to features in the UCC and thence to specific segments of code or constructs.

4.3 Levels of UCC and Consequence Graph

There is one last twist to the story of UCCs and consequence graphs, which is that there are levels of them that correspond to the levels of scheduler in a hierarchical machine. We use an example involving a server machine with a hierarchy of runtimes to illustrate both, concentrating first on just the UCCs, then adding the consequence graph.

4.3.1 Levels of UCC

For the example, consider a server with one rack, having a back-plane that boards plug into. A board has its own memory with four sockets, each having a chip with four cores. So there is a back-plane network connecting the boards, a bus on each board that connects the sockets to the DRAM, and inside the chip in each socket is a cache hierarchy that connects the cores.

The hardware is given a set of runtimes to match the hierarchy. Each network or bus has a runtime that schedules work onto the things connected below it. So the top runtime divides work among the boards, while a board's runtime divides work among the sockets, and a socket's runtime divides work among the cores.

To a runtime high up, each runtime below it looks like a complete machine. It schedules work-units to those machines, without knowing the internal details of how that machine is implemented. So the runtime at the top handles very large work-units that it schedules onto the runtimes on the boards. A board-level runtime divides up the work-unit it gets into smaller work-units, then schedules one onto each socket's runtime, and so on.

The application in this example has been written in a language that allows work to be divided. The toolchain inserted a manipulator that allows each runtime to divide up the work it is given into smaller work-units, such as via the DKU pattern[]. This pushed the UCC of the application all the way to the right on the unit axis.

So what does the concrete UCC produced during a run look like? Well, a unit is defined as the work resulting from one scheduling decision. Each runtime has its own scheduler, which means units are defined for each runtime. That in turn means that each runtime has its own concrete UCC!

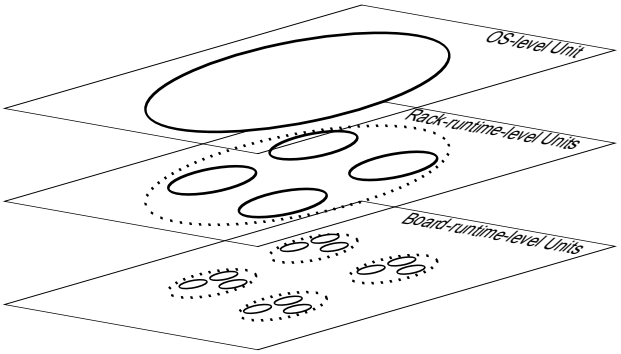


Figure 4: Representation of multiple levels of UCC.

Figure 4 shows that these UCCs are related to each other in the same hierarchy as the runtimes. A unit scheduled in one runtime is broken into smaller units in the one below it. Each of those units is then separately scheduled, making a separate UCC just for them. So, as the figure shows, a unit in one UCC has an entire UCC inside it.

Great, that makes sense, now what about the consequence graphs?

4.3.2 Levels of Consequence Graph

A consequence graph ties together scheduling decisions made on units with the consequences in the hardware of those decisions. But there are now multiple levels of consequence graph, one for each UCC. With multiple levels, a lower-level runtime is treated as a single "core" by the level above it. So, what does "consequence" mean in this case? The answer is that for performance tuning, the consequence of interest is the critical path.

That gives two goals: first, to get the consequences to the critical path, then second, to charge each segment of time on an actual physical core to exactly one box in one of the levels of consequence graph. We note that the critical path for one level is in terms of the work-times of its units, but those units now each have an entire consequence graph inside it. Hence the work time of a unit is the critical path time for the consequence graph inside it.

Now, the question is, which portions of the physical core time should get counted towards one higher-level unit? The answer is seen by looking at all the levels in the matrix multiply application, from the story in Section 3. Going up, the level above the execution under study involves the invocation of entire applications, via OS commands. At that level, a unit is an entire process, and the critical path of the SCG in Section 3 is the work-time of that unit. That leaves the time spent inside the OS as the runtime overhead assigned to that unit.

In the other direction, the lower level is the operation of the out-of-order pipelines in the cores, which have the equivalent of a runtime, in hardware. The hardware runtime consists of the dependency logic that determines which instructions are free, and the issue logic that determines which functional unit performs a free instruction. Hence, a unit is one instruction. The work time is the number of cycles it contributes to the critical path, due to dependencies limiting overlap. And the runtime overhead is the operation of the dependency and issue logic. Those don't contribute *directly* to the critical path, so instructions effectively have no runtime overhead.

We return now to the question of the time a higher-level unit uses the cores outside of its sub-units. Consider, in the matrix multiply code, the core usage spent while dividing the work and handing it to other cores. This is not work of the application, but overhead spent breaking the single application-unit into multiple sub-units. Even though it is in the application code, it's purpose is implementing the execution model, which makes it runtime overhead. But which runtime level? It's not part of the SSR language runtime, so not overhead of a unit inside the application, but rather it's for the application itself, as a unit! So, the core time spent calculating the division gets counted towards the application-level unit, while the time spent inside the SSR runtime creating the meta-units is counted towards those lower SSR-level units. But both are in the critical path, so both charged as work time of the higher-level unit.

Another way to view this is that by the process of elimination, the only core-time not accounted for elsewhere is the time spent dividing up a unit into smaller ones, and time spent accumulating the individual results back together. So this is what gets charged to the higher-level unit.

The last question is how to handle communication consequences, which result from decisions made in all levels? The decisions in higher-level runtimes set the context for decisions in lower-level ones, which links a higher-level choice to the consequences resulting from the lower-level choices. But the value of a consequence graph comes from linking the size of boxes in it to the decisions made by the scheduler. It's not clear how to divide-up the time cores spend waiting for non-overlapped communication, to assign portions to different levels. We have no good answer at the moment and leave it for future work.

5. Implementation

Now that the usage and theory are in hand, we give the details of implementation. We attempt a bottom-up view, to provide an alternate path to understanding the model and visualizations, as well as providing a path to derive the benefits directly from the details.

This section will show the level of effort needed to implement our approach for a new language. In short, this involves inserting collection points into the runtime, then modifying the post-processing that produces the visualization.

We cast the implementation in terms of the computation model, then identify the points inside the runtime that correspond to points in the model. It is only in these spots that instrumentation gets inserted into runtime code.

One benefit seen directly from these details is that many things remain the same across languages, because the computation model is invariant across languages. This includes the basic nature of the visualizations and details of the implementation process. The reason is that visualizations are based on units, constraints, and causality, all of which exist in every language (even though some older ones obfuscate units).

Another benefit evident from the details in this section is that the instrumentation is done only once, for a language. All applications written in the language inherit the visualizations, without any change to the application code.

5.1 Meta-units and unit life-line in the computation model

In preparation for mapping the model onto implementation details, we define a meta-unit and unit life-line. These form the basis for deciding points in the runtime where data is collected.

Every unit has a meta-unit that represents it in the runtime. A unit is defined as the trace of application code that exists between two scheduling decisions. Looking at this in more detail, every runtime has some form of internal bookkeeping state for a unit, used to track constraints on it and make decisions about when and where to execute. This exists even if that state is just a pointer to a function that sits in a queue. We call this bookkeeping state for a unit the meta-unit.

Each unit also has a life-line, which progresses so: creation of the meta-unit ➪, state updates that affect constraints on the unit ➪, the decision is made to animate the unit ➪, movement of the meta-unit plus data to physical resources that do the animation ➪, animation of the unit, which does the work ➪, communication of state-update, that unit has completed, and hardware is free ➪, constraint updates within runtime, possibly causing new meta-unit creations or freeing other meta-units to be chosen for animation. This repeats for each unit. Each step is part of the model.

Note a few implications: first, many activities internal to the runtime are part of a unit's life-line, and take place when only the meta-unit exists, before or after the work of the actual unit; second, communication that is internal to the runtime is part of the unit life-line, such as state updates; third, creation may be implied, such as in pthreads, or triggered such as in dataflow, or be by explicit command such as in StarSs, and once created, a meta-unit may languish before the unit it represents is free to be animated.

Also, note that this explains why the visualizations remain largely the same across languages. The concepts of a meta-unit, a unit, constraints on a unit, and a unit life-line are all valid in every language. The visualizations are based on these concepts, and so likewise largely remain the same. In the UCC, only the constraint patterns that represent the language's constructs change between languages. In the SCG, only which construct a line in the SCG represents changes.

5.2 Mapping model onto implementation details in runtime

The meta-unit and unit life-line aspects of the computation model map straight-forwardly to the UCC visualization. The constraints in the UCC are those stated in or implied by the application (with the complexities noted in Section 4).

However, the SCG is not a strict expression of the model, rather it's purpose is practical. It shows usage of the cores, and relates that to the quantities in the model. Hence, the measurements for the SCG all are boundaries, where the core's time switches from one category in the model to a different.

This differs from the model in subtle ways. Most notably, the model declares segments of time where communications take place, while the SCG doesn't measure the communication time directly, rather it captures idleness of the core caused by the non-overlapped portion of that communication.

This difference stems from the SCG's focus on core usage, and assigning each idle period to a cause. The runtime's choice of units to cores is what determined the source and destination of communications, which caused the idling. Hence, idle periods due to non-overlapped communication are consequences of the assignment choices made by the scheduler. This supports the name: scheduling consequence graph.

What must be collected during the run differs between the two types of visualization. For the UCC it is unit boundaries and the constraints related to each unit. For the SCG, the same units must be collected, but also the time a core spends on each segment of the unit's life-line. Also, implementation details of the runtime will cause things such as idling the core during lock acquisition to be counted towards a unit's life segment. What core activities go to which life segments changes from runtime to runtime. For example, our implementation includes idle time due to acquiring the lock on shared runtime state as part of the state-update life-line step.

The SCG represents each cause of a transition from one segment of core usage to another as an arc between boxes. Such a causation is always a causal dependency of some kind, because the SCG only represents physical events, even if it corresponds to a complex construct in the application. These causations are collected and tied to one of: construct constraint, runtime internal constraint (such as must acquire lock), or hardware constraint (such as only one activity at a time on a core). In this paper, all are collected, but the only causations visualized are constructs that cross cores, with propendent on one core and its dependent on another.

5.3 Instrumenting our implementation of SSR on top of VMS

We instrumented a version of SSR implemented on top of a proto-runtime system called VMS. This proto-runtime embodies most of a runtime implementation, but has replaced two key portions with interfaces. Those portions are the handling of language construct-constraints and the decision of which core to assign a unit to. To implement a language, one simply supplies those two portions of code, via the interface.

VMS also has the advantage for our approach of being written in accordance with the computation model, which makes instrumenting it especially convenient. Each language construct has its own handler into which to insert measurement code, and transitions in unit life-lines also have convenient locations in VMS to insert instrumentation code.

5.3.1 SSR background

A distinction important to understanding SSR and other parallel languages is being task-based versus virtual processor (VP) based. Task-based languages include dataflow, CnC, and StarSs. These tasks don't suspend and resume, but rather execute to completion. Hence, such a task is the same as our definition of unit. They have no state that persists across calls to the runtime. In contrast, a virtual processor does suspend and resume and so has state that persists across runtime calls. Examples include pthreads, OpenMP thread-based constructs, UPC, and so on.

SSR is based on virtual processors. They execute sequential code that occasionally calls a parallel construct, which suspends the VP and switches to the runtime. This means that each VP contains a sequence of units, with each unit the trace-segment between two SSR library calls.

SSR has both deterministic constraints, which specify the source and destination VP, such as send_from_to, and non-deterministic

ones, in which the runtime is what chooses which VPs interact, such as send_of_type_to and singleton. Deterministic ones display the same in the UCC and the SCG. However, non-deterministic ones need all possibilities to be determined for the UCC, requiring extra instrumentation code.

5.3.2 Collecting a unit

Code to record a new unit is inserted into VMS at the transition out of the runtime and into application code. Code to record the unit end is inserted into the VMS primitive that switches back to the runtime.

5.3.3 Collecting the constraints

In VMS, each language construct has its own handler. Code is inserted into each handler, to record which unit invoked the construct, and any units freed by it. The SCG links the unit that made a construct call to the units freed by that call.

What information needs to be collected for SCG and UCC and how it is done depends on the construct:

- create_VP: We place code into the create_VP handler, which records the calling VP + unit, along with the newly created unit, and the VP it is assigned to. Both the SCG and UCC draw arcs between creating unit and created.
- send_from_to and receive_from_to: Code is placed into both handlers at the point that checks if both the rendez-vous requests are present. When true, it records both the unit+VPs that connected. The UCC and SCG both represent this by two crossing dependencies.
- Send_to_of_type and receive_to_of_type: The same code is inserted to record both the unit+VPs that connected. This is enough for the SCG. But for the UCC, we want to capture all sending and receiving permutations available, so we add code that collects the group of senders and the corresponding group of receivers.
- Singleton: The singleton unit has a group of predecessor units and a group of successor units. The first predecessor to complete enables the singleton unit, while all successors must wait for its completion. We insert code into the handler, which records the predecessor that enabled the singleton. This is all that the SCG needs. For the UCC, we add code inside the singleton call that collects the calling unit, adding it to the predecessor group, and the unit it jumps to, adding that to the successor group.

5.3.4 Recording time, instructions, and cache misses

Just recording the units and connections between them is not enough. Because the SCG represents core usage, it also needs the cycles spent on each activity, including internal runtime activities. The size of each interval of core usage is recorded and assigned to a segment of a particular unit's life-line.

The UCC also makes use of the number of instructions in a unit, as an estimate of size of work in the unit, as illustrated by Fig [UCC same-sz vs UCC instr-sz]. Without knowing the relative size of the units, it is hard to estimate the amount of parallelism *usefully* available in the application.

To measure the instructions, cycles, and communication (cache misses), we use hardware performance counters. Readings are inserted into the runtime code to capture core time spent on each segment of the life-line of a unit:

1. Create meta-unit: This is the time spent inside the create_VP construct's handler function.
2. Update constraints: This is the time spent inside the handler functions that implement the constructs.
3. Decision to animate: This is the time spent inside the language-supplied assigner function.
4. Move meta-unit to core: This is via shared variables, recorded as part of 3.
5. Move work data to core: This is via cache misses, recorded as part of 6.
6. Do the work of the unit: This is the cycles between the switch-to-unit and the following switch-to-runtime.
7. Communicate state update: This is the time between leaving the application code and starting the construct handler (which includes lock acquisition).
8. Resulting constraint updates: This is the time spent inside the construct handler, and is the same as 2.

To cover each of those segments of a unit's life-line, code to read the performance counters is inserted at:

- Construct handler: To measure 2 and 8, reading is done before and after VMS calls the language-supplied construct handler function.
- Assigner: To measure 3 and 4, reading is done before and after VMS calls the language-supplied assigner function.
- Work: To measure 5 and 6, reading is done by reading inside the VMS switch-to-unit operation, and the switch-to-runtime operation.
- Dual-use: 1 is measured by using the construct handler reads for the create_VP construct handler. For 7, the switch-to-runtime read is subtracted from the read at the start of the construct handler function.

For clarity, all but work are grouped as overhead in the visualization, but they could be displayed separately if needed.

All the measurements are output into a trace file, which is then evaluated after the run to build the visualizations.

5.4 Building the Visualizations

Both the UCC and the SCG are internally represented as directed graphs, with units as nodes.

5.4.1 UCC

For the UCC, units can be either unweighted or weighted. Weighted units appear as rectangles with height proportional to the weight, unweighted units appear as circles. We weight the units with the number of instructions in the work. This removes some of the influence of scheduling and data, such as cache misses.

A critical path algorithm calculates vertical position of a unit, as its position within the critical path. The nodes are spread horizontally such that none overlap.

Simple constraints (dependencies) are painted as arcs. Complicated constraints are for now displayed as an additional node bearing information on the constraint, with incoming arcs from all units whose execution status affects the constraint and outgoing arcs to the constrained units.

5.4.2 SCG

For the SCG, all nodes are weighted with the number of cycles spent on the unit in total (work + overhead). For display, the nodes are split into overhead and work. The same critical path algorithm as for the UCC is used to place nodes vertically, but this time horizontal placement is determined by the core on which the unit was executed (hardware dependencies ensure no overlap).

Constraints can then be overlaid, color coded by type. By default, in SSR, we display creation, direct and typed message send-

ing (but not the crossing dependency from the receiver back to the sender), and singleton outgoing dependencies; but each type can be individually hidden or shown.

All this information is taken purely from the runtime, leading to a rich, configurable visualization without needing to add anything to the application.

6. Conclusion

We have shown how to apply a computation model to instrument a language runtime for collecting measurements that connect: each measurement to others, to application structure, to scheduling decisions, and to hardware. A simple visualization of the data has features that indicate lost performance, and features that visually link the lost performance to the cause, no matter if the cause is application structure, language runtime implementation, or hardware feature. It is this linkage, due to the computation model, that sets this approach apart from others.