

---

**Pages**

Hans Hagen, Mikael P. Sundqvist

**1 Introduction**

When we discussed the par builder<sup>1</sup> we went into the details about how  $\TeX$  tries to produce the best paragraph possible. We discussed the traditional process as well as some extensions. We also discussed the possibility of defining multiple passes driven by various parameters.

Eventually paragraphs make pages by means of the page builder. In comparison to the par builder, the page builder is relatively simple. There is only one pass and it's not even a pass; when something is added to the main vertical list the builder can decide if the conditions are such that a page is split off. Where the par builder looks at the whole and checks various solutions, the page builder is a forward-looking process. It remembers the last acceptable break and will use that one when a new contribution doesn't fit the conditions, that is: overshoot. The actual break routine is small and the complication is mostly in the way inserts are handled (think footnotes and floats). A simple version of the page builder is also available as a primitive: `\vsplit`.

When we want to typeset a document in columns, and also want to support inserts, we have to involve the page builder. Say that we have two columns. A simple approach is to typeset on a narrow page and keep the first column around and package that with the second one when the time is there to ship out the result. If we also want to balance the last page, we need to collect up to twice the column height and use `\vsplit` on the output box. Some juggling is needed to handle inserts right.

In  $\text{Con}\TeX$ t we have several column mechanisms and each has its advantages and drawbacks. One is used for mixed single- and multicolumns. Another is exclusively for multi-columns and supports features that are otherwise not possible (or at least hard to do). There are other box-oriented column handlers. There is also so-called 'column sets', a mechanism that targets magazine-like layouts. It supports explicit placements of elements, and dates from MkII; it has been converted to MkIV, but we can do better in MkXL. It is this mechanism that we (Mikael and Hans) decided to improve.

Column sets are grid-based and quite flexible. They can have a different number of columns on the left and right pages. The columns can have

different widths when we do parallel text streams. The number of lines and the start can be different per page. Elements can span a spread and images can be placed in various ways, for instance 'try to place this graphic from the right bottom upwards and move to a previous column when there is no space' also known as `btr1` or 'bottom top right left'. There are various additional placement features.

In the MkII and MkIV variant we mix placement of graphics with content but in MkXL we also provide a way to do the placement beforehand. This is more reliable because we know that there is no content yet, so we can go back and forward. Because in that approach we know that we have only text, we thought that it would be nice if we could optimize the distribution of content over the available space. For that we added two features to  $\text{LuaMeta}\TeX$ : main vertical list intercepting and balancing.

**2 The main vertical list**

Content is eventually collected in vertical lists. That can be a `vbox` or equivalent, or it can be what is called the main vertical list or `mvl`. When something is added to that, some `mvl`-specific checks are done. Initially the `\vsize` (register) is used to initialize the `\pagegoal` (global state property) and successive contributions update the `\pagetotal` (also a property). Inserts are taken into account and they can bring the goal down. When the page overflows, the `\output` routine (an internal token list register) is expanded after the content has been packaged in a vertical box, traditionally `box` register 255, but that number can be changed in  $\text{Lua}\TeX$ .

Two columns are basically two times the vertical height, so you can set the target twice as high and let the output routine decide when to split it. One complication is that there can be too much, and then the left-over content, a.k.a. discards, has to be 'pushed back' into the input ( $\text{Lua}\TeX$ 's `\pagediscards`). Although one can also push back what  $\TeX$  decided to throw away (like glue at a boundary), this is rather fragile and one can lose information.

It is also hard to ask at any given point what the exact state is, because who knows what is pending or yet coming (like `\parskip`). The goal and total are not always reliable (although in  $\text{LuaMeta}\TeX$  we have additional variables that can be queried). All this makes multi-column emulations in what is essentially a single-column engine somewhat hard. Even more tricky is proper balancing of columns, which boils down to a loop using `\vsplit`.

Although  $\TeX$  became popular rather quickly for typesetting journals, its single column approach had to be compensated for by macro-based solutions,

---

<sup>1</sup> A new take on paragraphs, Hans Hagen and Mikael Sundqvist, *TUGboat* 46:1, pp. 104–123. [tug.org/TUGboat/tb46-1/tb142hagen-paragraphs.pdf](http://tug.org/TUGboat/tb46-1/tb142hagen-paragraphs.pdf)

and those are often rather complex because they need to deal with ‘all’ possible cases. In *The T<sub>E</sub>Xbook* we find solutions that likely match the intended use of the splitter: well-defined input that when systematically used can drive a relatively simple split routine. As with many things relating to T<sub>E</sub>X, it is arbitrary unpredictable input that complicates solutions.

These complications are one of the reasons why we called the new multi-page routine the balancer: it needs to find an optimal solution for breaking into what become columns and also be able to do proper balancing. But we deliberately put some constraints on this, as we will see. Of course we can also use the balancer to get a single column (multi-page) solution because the number of columns is no longer the issue here. Internally we talk of filling up slots; how these are finally assembled doesn’t matter.

### 3 The balancing act

When we look at what we call the balancer we should go back to the par builder because that is what we started from. We asked ourselves what were the similarities and what could we borrow. As a starter: think of lines as words. Let’s look at what the par builder does:

What goes in is a list of nodes (think glyphs), kerns, glue, penalties, boxes, whatsits, boundaries, rules, discretionaries, and more. How that breaks into lines is determined by where one can break, for instance at glue or penalties, or within a discretionary (like a hyphenated word). But the process is more complicated as we have a lot to consider:

- orphan penalties (lone words on the last line), toddler penalties (single glyphs at the margin) and twin penalties (similar words at the margins)
- spacing glue with optional stretch and shrink (between words), punctuation glue (after punctuation) and emergency stretch (just in case we need more wiggle room)
- left, middle and right boxes (repeated content at breaks), leaders (adaptive content), inline math (with specific controls and constraints), and optional content (multi-pass controlled)
- glyph expansion (stretching and shrinking), character protrusion (escaping into margins)
- discretionaries (end and begin line snippets) and various hyphenations (breaks within words), both with optional penalties
- per-line shape control with carry over and/or repeat, begin paragraph indentation, hanging indentation from one of the four corners (width-related)

- left and right margins, left and right hanging, initial left and right skips, final left and right skips (all determine line width)

Most of these concepts, some of which are unique to LuaMetaT<sub>E</sub>X, sort of translate into similar concepts for a page builder. But before we come to that, we identify what the parbuilder delivers to a vertical list, namely lines, but separated by for instance:

- widow penalties (between last lines)
- club penalties (between first lines)
- shape penalties (between shape lines)
- broken penalties (between hyphenated lines)
- interline penalties (between lines)

These are related to line breaks in the sense that the routine inserts them when lines get contributed to the vertical list. They are often discussed in the context of par building but effectively kick in when a ‘page’ is built. They are present in a `\vbox` too, but there they have an effect only when a packaged box is unboxed and contributed to the page.

So the above penalties we need to handle anyway, and we can try to translate the (spacing and width) concepts into similar concepts of building pages. There are however, yet more aspects, and these cross the boundary between par and page:

- marks (like running header states) between last lines
- inserts (like footnotes or figures) between first lines
- vadjusted material (vertical material) before or after a line

These are special because in traditional T<sub>E</sub>X they have to ‘migrate’ out of a line into the vertical list. In LuaMetaT<sub>E</sub>X that also happens when they are what is called ‘deeply buried’ inside boxes. By the time we can see them in page breaking they are there but considered invisible, while inserts are part of the flow and their properties (dimensions, etc.) influence the break points. Adjusts are already injected and have become regular vertical items.

So, in the end there are many possible aspects to consider, and much of the above translates. Just try to make up a list of demands before reading on; here is a clue: discretionaries make little sense but we need to bring back something similar. Here we go.

The page builder gets lines. These are equivalent to words. Although one can envision a discretionary approach we decided to drop that idea because there is no easy way to define an interface for this in practice. It’s not like users write texts with alternative lines or even snippets of lines. So, if we start from lines, and boxes otherwise injected, building a page involves this:

- `vsize`: the main constraint as it should not overflow
- `inserts`: these are bound to preceding content but can move and split
- `glue`: it can be flexible and enlarge the solution space or be fixed (grid) and impose a limitation
- `penalties`: these drive the routine but relative values demand care
- `adjusted content`: this already is part of the list but can interfere
- `marks`: these are kept with content and interpreted when packaging happens
- `vz`: is vertical stretch comparable to micro-typographic font expansion, a.k.a. ‘hz’, applied after the packaging operation
- `top and bottom skip`: this has to be applied when initializing a ‘page’ and taken into account
- `extra goal`: this provides some optional wiggle room
- `feedback loop`: trial and final runs permit column construction; this includes control over and access to marks, inserts, etc.

We don’t mention additional constraints here, for instance spacing-related tweaks to deal with half-line spacing when typesetting on the grid, snapping on the grid of content, enforcing a column, page or spread break, etc. An engine can help here but these are rather macro package-specific constraints.

The feedback loop is triggered by output penalties and a so-called output routine can then take what is there, deal with it, or push it back and continue. As we already mentioned, this is a progressive process: essentially, we don’t look back. One complication is that when the output routine is triggered, we don’t know the dimensions involved because that state has been obfuscated by the fact that we went beyond a threshold. It means that one might end up with repacking unpacked content to get the real state of affairs. We already extended `LuaMetaTeX` to provide more reliable feedback but that is not enough to reach our goals. It doesn’t help balancing.

So, in order to get through our wish list, while keeping all of the above in mind, we decided to start from what in `ConTeXt` are called column sets: a mechanism for creating multi-column pages in a magazine-like way, with for instance graphics spanning columns or spreads, and more precise control over placement. How does that compare to other column mechanisms? Equally important is the question what kind of support we need from the engine. This is what we provide in `MkIV`; we mention it so that you notice the different approaches:

- `Simple columns`: Collect content in a `vbox` and let `TeX` do the splitting. It’s the quick and dirty fixed-placement approach.
- `Mixed columns`: Set the `vsize` to  $n$  times the text height and when we have a full page, split the collected content. This is sensitive for spacing and decisions around page and column breaks. The splitting is mostly delegated to Lua. What we call boxed columns are a variant of this; they are just objects. Figures are handled as on single column pages. The name ‘mixed’ indicates that we can have them in the middle of single column layouts, so for instance itemized lists and tabular content obey them.
- `Page columns`: Set the `vsize` to the text height and collect pages as usual but delay flushing till the number of columns is reached. The splitting is mostly done in `TeX` using the regular builder. The advantage of this method is that all single column mechanisms work as expected, including side floats. However balancing is not part of the deal (hackery at best).

- `Column sets (old)`: Split the page into blocks (slots) where content can go. Areas on the grid can be reserved, blocked, or occupied by, e.g., figures. When set up, change the `vsize` every time a slot (pseudo-page) is split off and stepwise populate the page. It also works on spreads. In `MkIV` we let Lua do some of the work.

This is the new approach:

- `Column sets (new)`: Define a sequence of page and spread templates. Put fixed material in specific locations; to some extent that can be done as part of the flow, but it’s less predictable. Collect content and at some point flush what has been accumulated to fit those templates. This is done in multiple passes, until a good solution is found, much like the `par` builder does with words.

This last is what we’re after and by comparison is rather different from the others. Those basically use the regular page builder and in practice do what we expect them to do quite well. It’s the column sets that demand something better. For the record, the traditional `TeX` engines have both a page builder and `vsplitter`: there are some similarities but the `\vsplit` has its own code. Both of these mechanisms have been extended a bit in `LuaMetaTeX`, but the third method, the balancer, is independent of these.

- We started with a similar approach as the `par` builder: choosing the best solution out of (possibly) many.

- We used the par builder routine but dropped features specific to a running text. For a while we kept discretionary code but dropped that too. We also didn't need the multitude of penalties used there.
- What did make sense was exploring page shapes analogue to par shapes: heights (`noflines`), top-skip and bottomskip conditions kick in. Shape entries relate to what we call slots in columnsets. There one sees some similarities.
- That set us on the path of trial runs and adaptive shapes. However, this is done at the  $\TeX$  end in loops that can test conditions and results and retry.
- At first we didn't want to do inserts, but then realized that it was doable given some constraints: they are slot bound; end notes migrate to the end, page notes end up at the bottom, and margin notes end up alongside.
- Because we want to work on the grid, we had to take care of half-line spacing and synchronize at breaks. This introduced top and bottom optional content, basically dumb discretionaries but done differently.
- In the end we also realized that trial runs made it possible to implement high performance balancing.

So all in all we see similarities, most noticeably the shapes, but although for instance left and right skip translate into top and bottom skip, in the end they demand different code, because they also interact with what goes on top or at the bottom: they can be discardable too. The concept of hyphens (discretionaries) can be seen back in discardable content, but where in the par builder this is taken into account explicitly, in the balancer we cheat a bit. In the par builder we have protrusion, expansion, left and right boxes that all determine the break points. One way the builder decides to reject or accepts a solution is in the under- or overfull boxes, where it is more tolerant of the latter. In the balancer we therefore conveniently used the concept of overfull boxes and hooked for instance discardable as well as top and bottom skips into that place. It means that we can get overfull (so-called) slots, and those can be acted upon if needed. This is not different from the par builder happily accepting some overfull lines when no better solution is seen.

Another aspect is inserts. Here we could try the somewhat complicated approach in the page builder (it makes for most of the code there) but easier is to just temporarily put the inserts in the list as content and then collect it later when we use a filled slot. So,

in the end, when we compare with the par builder the similarities are kept: build a solution tree and come up with a solution in multiple steps. That brings us to passes: one can optimize the balancing with multiple passes, but we need to experiment a bit more with this. We just mention here some of the variables to play with in the par builder:

- `threshold`
- `tolerance`
- `looseness`
- `adjdemerits`
- `originalstretch`
- `emergencystretch`
- `emergencyfactor`
- `emergencypercentage`

We have less than that in the par builder but still enough to experiment with. It is time to show the ingredients that the engine provides to implement this.

#### 4 How it works

We will explain what mechanisms we have and how they can be used. We start with intercepting the main vertical list, the page stream:

```
\beginmvl 1
  various content
\endmvl

\setbox\scratchboxone\flushmvl 1

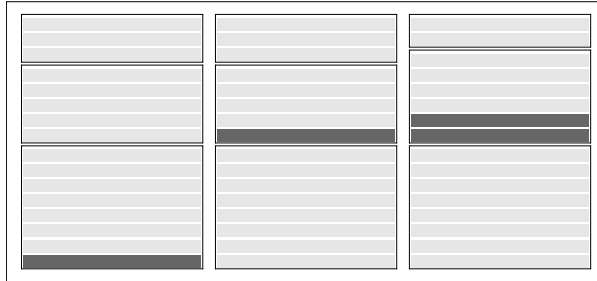
\beginmvl
  index 2
  % prevdepth 4pt
  options \numexpr "1 + "4\relax
\relax
  various content
\endmvl
```

There are some options (with Con $\TeX$ t names for bits):

```
0x1 ignore prevdepth \ignoreprevdepthmvloptioncode
0x2 no prevdepth \noprevdepthmvloptioncode
0x4 discard top \discardtopmvloptioncode
0x8 discard bottom \discardbottommvloptioncode
```

When all content is collected we can create a sequence of slots defined by a so-called balancing shape (using some Con $\TeX$ t variables):

```
\balanceshape 3
  vsize 3\lineheight % 3lh
  topskip \strutht % 1sh
  bottomskip \strutdp % 1sd
next
vsize 5\lineheight
topskip \strutht
```



**Figure 1:** There can be multiple possible solutions. The last one overflows in a not-shown next slot.

```

bottomskip \strutdp
next
vsize      8\lineheight
topskip    \strutht
bottomskip \strutdp
\relax

```

Balancing is then called for explicitly, and the solution depends on the conditions and content (see figure 1).

```

\setbox\scratchboxtwo
\vbalance\scratchboxone

```

When we are satisfied the results are flushed in an endless loop (one of these new native LuaMetaTeX features) that we quit when we're done:

```

\hbox \bgroup
\localcontrolledendless {%
\ifvoid\scratchboxtwo
\expandafter\quitloop
\else
\setbox\scratchbox
\ruledhbox\bgroup
\vbancedbox\scratchboxtwo
\egroup
\ vbox to 12\lineheight \bgroup
\box\scratchbox
\vfill
\egroup
\hskip1em
\fi
}\unskip
\egroup

```

Normally you will implement trial loops because as we saw in the picture we can have an overflow (here `trial` is a keyword):

```

\setbox\scratchboxtwo
\vbalance\scratchboxone trial

```

A trial run gives empty slots and one can check the dimensions and decide to adapt the page shape accordingly. For instance, if a slot overfills then that slot can get one line less. Often, a few iterations give an okay result. The trial runs are quite fast because the number of lines on a page is not that large, so distributing 50 pages of content over a few hundred

slots takes little time. Keep in mind that paragraphs have already been built and figures are already placed. Setting up a huge page shape can be done efficiently in Lua. As they are temporarily placed in the flow, handling inserts takes no noticeable runtime either.

You can force a break with a boundary (these become nodes). Using these with a proper callback we can go to a next slot (there can be more in one column), column, page or spread.

```

\balanceboundary 3 1\relax
\vskip\zeropoint
\balanceboundary 3 0\relax
\vskip\zeropoint
\balanceboundary 3 0\relax

```

In ConTeXt for instance we implement these:

first	second	action	user interface
1	1 or 0	goto next spread (1 initial, 0 follow up)	<code>\page[spread]</code>
2	1 or 0	goto next page (idem)	<code>\page</code>
3	1 or 0	goto next column (idem)	<code>\column</code>
4	1 or 0	goto next slot (idem)	<code>\column[slot]</code>
5	n	next slot when more than n lines	<code>\testroom[5]</code>
6	s	next slot when more than s scaled points	<code>\testroom[80pt]</code>

Marks are kind of hidden in the resulting (split off) segments of the balanced list. They can be set using Lua calls:

```

node.direct.updatetopmarks()
node.direct.updatefirstmarks()
and
node.direct.updatemarks(list)
node.direct.updatefirstandbotmarks(list)

```

The pattern is:

- set the top marks
  - loop over columns and set marks
  - set the first marks
- How about inserts?
- We can assume a reasonable amount of notes.
  - These are normally small with no (vertical) whitespace.
  - Notes taking multiple lines may split.
  - But we need to obey widow and club penalties.
  - There can be math formulas but mostly inline.
  - We need to keep them close to where they are referred from.
  - We can ignore complex conflicting demands.
  - As long as we get some result, we're fine.
  - So users have to check what comes out.
  - We don't assume fully automated unattended usage.

One can check for inserts with

```
<state> = \boxinserts <box>
```

and fetch them with

```
<box> = \vbalencedinsert <box> <class>
```

Possible states are: ‘has inserts’, ‘has inserts with content’, ‘has inserts with height’.

We need to deal with (for instance) half line spacing on the grid which is a bit hackish but also sort of generic. Kind of like this:

We place a discardable rule that triggers something:

```
\vskipOpt \hrule discardable
  height 1sh depth 1sd width 1em \par
```

We define something optional before:

```
\vskipOpt \vbox discardable
  {\hpack{\strut BEFORE}} \par
```

And some after:

```
\vskipOpt \vbox discardable
  {\hpack{\strut AFTER}} \penalty -1 \par
```

In the end, left over discardables are turned into something else:

```
% becomes a \nohrule:
\vbalenceddiscard 2 \relax
% disappears:
\vbalenceddiscard 2 remove\relax
```

This mechanism is still evolving but seems to work well already. Because in column sets we work on a grid, discardables are also used for making sure that for instance half-line spacing around display formulas works out well. The most complicated issues occur at the top of columns because there we also interfere with topskip (a concept that we’re rather stuck with, as it is part of T<sub>E</sub>X’s design).

It is possible to discourage breaks at the end of a list to be balanced, like this:

```
\balancefinalpenalties 6
  10000 9000 8000 7000 6000 5000 \relax
```

There is more to tell but we leave it at this. In the column set application the amount of code involved in this is relatively small. Managing the process and specific features are done in Lua; for instance there we populate the shape there. Keep in mind that we also need to manage figure placement and flushing, bind them to slots (columns and pages), support columns of different heights, all things that also were in the old column sets. In the end we can conclude that most time went not so much into writing the code but wrapping it into a user interface that we find intuitive: it was an example of a project that took a bit of time but also discussing and testing on a daily basis for quite a while. It’s how we like development and we have a lot of fun doing it that way.

## 5 Applications

In the ConT<sub>E</sub>Xt distribution one can find use cases. We tested all with Mikael’s lecture notes. These contain lots of graphics that can span columns, are positioned in specific places and mix in math. They are about as complex as one wants to go, but we focused on easy input. So in the end it boils down to defining images and placement, entering content and hoping for the best. One set of notes (a chapter) is typeset in one go and when specified well, not much extra work is needed. You get what you ask for. The presentation given at BachoT<sub>E</sub>X and the ConT<sub>E</sub>Xt meetings in 2025 is made with the new columnset module and shows several examples of how text and graphics can be distributed over (in that case) up to three columns. (See next page.)

The distribution also has an example of parallel typeset bibles. We take publicly available XML files, map elements on commands and can get three different variants: parallel chapters, parallel verses and highlighted differences between translations. In the first two cases processing for instance four parallel translations gives some 3000 pages but processing still stays within minutes, which is satisfying given the amount of work to be done. The file `m-bibles.mkx1` supports (by default four) parallel versions over a spread (but you can of course also go for another assembly) where you can synchronize chapters, verses or show differences. Showing an example here makes little sense because it would be too small; just process the file.

We’re not yet finished but hope to find time to do so. As with other fundamental extensions to the engine (math, par builder, etc.) these things are rewarding, which makes them being demanding in terms of time not much of an issue. And they make for nice presentations and discussions at meetings. After all, most of working with T<sub>E</sub>X is about the results.

- ◇ Hans Hagen  
Pragma ADE
- ◇ Mikael P. Sundqvist  
Department of Mathematics  
Lund University

