overload 111

OCTOBER 2012 £3

Software Developer Business Patterns

A series of "business patterns" specifically tailored to software developers

Keep It Simple, Singleton!

We see the advantages of "naively" mapping business tasks into software design

A DSEL for Addressing the Problems Posed by Parallel Architectures

Using a domain specific language to avoid deadlocks in parallel code

Valgrind: Cachegrind and Callgrind

We continue to investigate the more advanced Valgrind diagnostic facilities

Universal References in C++11

Scott Meyers demonstrates the new C++ facilities for referring to objects

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 111

October 2012

ISSN 1354-3172

Overload is a publication of ACCU
For details of ACCU, our publications and activities, visit the ACCU website: www.accu.org

Copy deadlines

All articles intended for publication in Overload 112 should be submitted by 1st November 2012 and for Overload 113 by 1st January 2013.

ACCU

ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

The articles in this magazine have all been written by ACCU members - by programmers, for programmers - and have been contributed free of charge.

Too Much Information

Overload usually has an editorial. Frances Buontempo explains why she hasn't had time to write one for this issue.

I apologise that I have not had time to write a proper editorial this time. I have been far too busy, but I'm not sure what I've spent my time doing, so I took some time out to research this. Hopefully the results will suffice, in lieu of an editorial.

How do you spend your time? How much of it is in front of a computer? How much of it is dealing with emails or being distracted, or even informed, by social networking sites? My personal answers are

1. Unwisely
2. Too much

I wondered if I spend too much time on emails because I get more than some people, so asked on accu-general. I got four numerical answers, so this is clearly not valid to draw general conclusions from. Nonetheless,

Paul F:

> For my summer hols, I didn't read any e-mail for 3 weeks [including 1 day bank holiday], and I had about 1000 mails or about 70 per working day at work. On this account I had something like 1600, almost all from mailing lists (accu-general, Qt interest, doxygen, valgrind) or spam, or about 80 per day, working or not working.

Alan G:

> Yesterday I was offline and so got 0. On Wednesday I got 258 (plus some on accounts that are less easy to count).

Richard Howells:

> About 100 that I actually see, plus about 100 the spam filter catches. I skim the list and occasionally notice the odd false positive. There may be more false positives that I don't notice.

Colin Paul Gloster:

> After deleting (some) spam I estimate an arithmetic mean of circa 147 emails per day in a sample of 243 recent days.

Previously, my gmail account got about 100 a day, ignoring spam. This has now dropped to 20 or so, because my email now automatically marks certain messages as read and sends them to a relevant folder. This allows me to use my inbox as a to-do list and choose when to spend time reading things on discussion groups etc. However, I do still spend too much time reading them. The problem of deciding what to read in detail also applies to facebook and twitter. Certainly twitter messages are short, but sometimes twitter reaches a speed of one tweet per minute. Several of these also end up on facebook. Oh for something to just summarise this for me without the duplication, and also filter out 'noise', such as "Aaaaa-waaaaaaaaaaaaaayyyyyyyyyyyy!", "phew", "Nooooooooooooooo!", "Home again, home again. (Still quite pissed)".[1] This data deluge is certainly a case of far too much information.

The article 'Am I wasting my time organizing email?' suggests "foldering may be a reaction to incoming message volume" [Whittaker11]. Most people on the accu-general 'survey' mentioned the spam filter. This most basic form of folder is extremely common. Various algorithms are employed to catch spam. I wonder if similar algorithms could be employed to catch other emails, that while not spam, the recipient would have no interest in. As Alan Griffiths mentioned, deciding what to read can be influenced by mood, so trying to automate this could be difficult.

How do people decide which emails to read?

Paul F mentioned glancing at automatic status updates just to check there are no problems. I have used filters to send automatic updates to folders, attempting to leave them as unread if there's a problem. This reduces the volume of email to deal with. Why are there so many machines sending out emails that other machines 'read' and then squirrel away? Perhaps there is no point in emails such as these. Since we are also influenced by who sent them, possibly the subject line, possibly how much time we have on our hands, attempting to automate an 'Is it worth reading' filter would be an interesting challenge.

Why do we save emails?

A blog posting on the 37signals website [37signals05] asks the question, do we save our phone calls. No we don't. We might take notes if needed, but most phone call conversations are throw away. We'll ring back if we forget something, so why do we keep such a huge volume of emails? Upon asking Google, 'why do we save emails', I was presented with information on:

1. Why I need to save money
2. Why we need to save the Arctic
3. Why I need to save rooster tail feathers (that one nearly pulled me in, wasting more time)
4. Why should we save tigers.

1. You know who you are!

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She has been a programmer for over 12 years professionally, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

Since Google won't answer the question this requires some thought. Some employers enforce email keeping, perhaps for regulatory purposes, but they will then archive all the emails for us. Having saved all the emails, how do we approach finding them later on? Einstein reminds us [Einstein]:

> Intelligence is not the ability to store information, but to know where to find it.

The re-finding email survey mentioned previously, [Whittaker11], concluded people's behaviour varied and was influenced by their email client: threaded conversations reduce the amount of scrolling down needed to scan for a previous communication, so tended to reduce the number of folders used. Having many folders requires a degree of effort and memory in order to find where you might have put something, and finally better searching tools speeds up retrieval. I sometimes wonder if anything really bad would happen if I deleted all my emails.

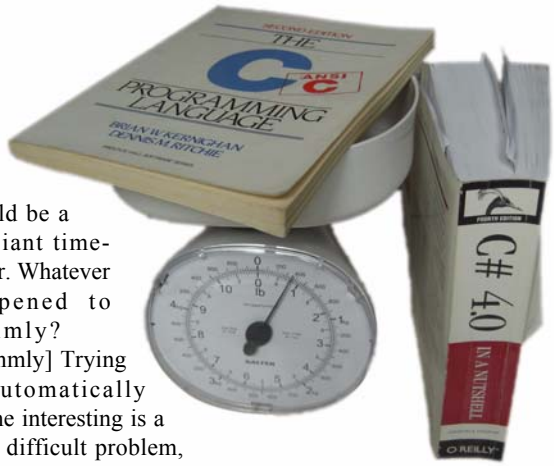
Perhaps Knuth's approach to email is the only sensible one [Knuth]:

> I have been a happy man ever since January 1, 1990, when I no longer had an email address. I'd used email since about 1975, and it seems to me that 15 years of email is plenty for one lifetime.

Though I have tended to concentrate on emails so far, facebook and twitter have been mentioned in passing. It sometimes seems that I am drowning in noise, but there are gems in the stream which I don't want to miss. Other things add to the stream of data. Why do I have so many bookmarks in my browser? In the process of researching something, I will leave myself a trail of other, sometimes only vaguely related, background to follow up on one day. Will I ever read them? Do they even exist anymore? Mind you, why do I have so many books in my house that I haven't read yet? Partly because I waste time trying to deal with emails, and maybe because I start reading books that are too big. Why are so many technical books so big? The other day, I weighed K&R. It come in at a sensible 375 grammes. How many technical books do you own that weigh less than this? How many books would you not put on your kitchen scales since they look far too big? I recently contacted Amazon and asked them to state the weight of the books they sell and allow me to order search results by weight. Feel free to back me up on this.

What have we learnt?

Trying to sift the wheat from the chaff manually is time consuming. Ideally something that would automatically summarise all interesting emails, tweets, facebook posts, books and so on and provide it in one short post would be a brilliant time-saver. Whatever happened to summly? [Summly] Trying to automatically define interesting is a very difficult problem, and might just be a day-dream. I have saved myself time by keeping an eye on the clock and being more aggressive about ignoring or deleting posts before being distracted by them.



References

[37signals05] 'Why do we treat email differently than a phone call?' (2005): http://37signals.com/svn/archives2/why_do_we_treat_email_differently_than_a_phone_call.php

[Einstein] Einstein – http://www.phnet.fi/public/mamaa1/einstein.htm

[Knuth] Knuth, D. 'Email (let's drop the hyphen)' http://www-cs-faculty.stanford.edu/~uno/email.html

[Summly] http://www.summly.com/

[Whittaker11] Whittaker, S., Matthews, T., Cerruti, J., Badenes, H. and Tang, J. (2011) 'Am I wasting my time organizing email? A study of email refinding' in *Proceedings of the 2011 annual conference on Human factors in computing systems*, also available from http://people.ucsc.edu/~swhittak/papers/chi2011_refinding_email_camera_ready.pdf

Valgrind Part 4 Cachegrind and Callgrind

When your application is slow, you need a profiler. Paul Floyd shows us how callgrind and cachegrind can help.

The good news is that you've read my previous two articles on Valgrind's memcheck tool, and now your application has no memory faults or leaks. The bad news is that your application is too slow and your customers are complaining. What can you do? There are plenty of options, like get faster hardware, better architecture/design of the application, parallelization and 'code optimization'. Before you do anything like that, you should profile your application. You can use the cachegrind component of Valgrind to do this.

Let's take a step back now for a very brief overview of profiling techniques. Profiling comes in several different flavours. It can be intrusive (as in you need to modify your code) or unintrusive (no modification needed). Intrusive profiling is usually the most efficient, if you know where to profile. The catch is that you usually don't know where to look. Profiling can use instrumented code, either instrumented at compile or link time or on the fly (which is the case for cachegrind and callgrind). Lastly, the profiler can be hardware based, for instance using the performance counters that are a part of Intel CPUs. as used by Intel Vtune. In addition to performing time based profiling, you can also perform memory profiling (for instance, using Valgrind's massif), I/O profiling and so on.

Profiling should give a clear picture as to whether there are any significant bottlenecks in your code. If you see that one function is taking up 60% of the time, then it's a prime candidate for optimization. On the other hand, if you see that no function uses more than a few percent of the time, then you're probably going to need to look at higher level approaches than code optimization (or else you will have to optimize a lot of code to make a big difference).

In general, profiling tools will tell you the time spent in each function, inclusive and exclusive of time spent in calls to other functions. They may also tell you the time spent on each line of code. Callgrind and Cachegrind generate output that has a lot in common, like 'time' spent per function and line. The main differences are that Callgrind has more information about the callstack whilst cachegrind gives more information about cache hit rates. When you're using these tools, you're likely to want to use the GUI that is available to browse the results, KCachegrind. This is not part of Valgrind; as the name implies, it is part of KDE.

Cachegrind

Let's start with a small example (see Listing 1). It's deliberately bad, and as already noted above, a classic case for using a better algorithm.

Compiling and running this is easy, just use Valgrind with the `--tool=cachegrind` option.

```
$ g++ -g badprime.cpp -o badprime
$ valgrind --tool=cachegrind --log-file=cg.out
  ./badprime
```

Paul Floyd has been writing software, mostly in C++ and C, for over 20 years. He lives near Grenoble, on the edge of the French Alps, and works for Mentor Graphics developing a mixed signal circuit simulator. He can be contacted at pjfloyd@wanadoo.fr.

```
#include <iostream>
#include <cmath>

bool isPrime(int x)
{
  int limit = std::sqrt(x);
  for (int i = 2; i <= limit; ++i)
  {
    if (x % i == 0)
    {
      return false;
    }
  }
  return true;
}

int main()
{
  int primeCount = 0;
  for (int i = 0; i < 1000000; ++i)
  {
    if (isPrime(i))
    {
      ++primeCount;
    }
  }
}
```

Listing 1

I measured the time it took to run (in VirtualBox on a 2.8GHz Opteron that's about 5 years old). Without cachegrind it took 1.3s. Under cachegrind that rose to 12.6s.

So what do we have. Well, I told Valgrind to send the output to `cg.out`. Let's take a look at that (Figure 1).

Some of this looks a bit familiar. There's the usual copyright header and the column of `== <pid> ==` on the left. The part that is specific to cachegrind is the summary of overall performance, counting instruction reads (I refs), 1st level instruction cache misses (I1 misses), last level cache instruction misses (Lli), data reads (D refs), 1st level data cache misses (D1 misses), last level cache data misses (Lld misses) and finally a summary of the last level cache accesses. The information for the data activity is split into read and write parts (no writes for instructions, thankfully!). Why 'last level' and not 'second level'? Valgrind uses a virtual machine, VEX, to execute the application that is being profiled. This is an abstraction of a real CPU, and it uses 2 levels of cache. If your physical CPU has 3 levels of cache, cachegrind will simulate the third level of cache with its second level.

There's a further option that you can add to get a bit more information, `--branch-sim=yes`. If I add that option, then (in addition to adding another second to the run time) there are a couple more lines in the output (Figure 2).

profiling tools will tell you the time spent in each function, inclusive and exclusive of time spent in calls to other functions

```
==1842== Cachegrind, a cache and branch-prediction profiler
==1842== Copyright (C) 2002-2011, and GNU GPL'd, by Nicholas Nethercote et al.
==1842== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==1842== Command: ./badprime
==1842== Parent PID: 1758
==1842==
==1842==
==1842== I   refs:      922,466,825
==1842== I1  misses:          1,234
==1842== LLi misses:          1,183
==1842== I1  miss rate:        0.00%
==1842== LLi miss rate:        0.00%
==1842==
==1842== D   refs:      359,460,248  (349,345,693 rd   + 10,114,555 wr)
==1842== D1  misses:          9,112  (      7,557 rd   +      1,555 wr)
==1842== LLd misses:          6,316  (      5,119 rd   +      1,197 wr)
==1842== D1  miss rate:         0.0% (        0.0%     +        0.0%  )
==1842== LLd miss rate:         0.0% (        0.0%     +        0.0%  )
==1842==
==1842== LL refs:            10,346  (      8,791 rd   +      1,555 wr)
==1842== LL misses:           7,499  (      6,302 rd   +      1,197 wr)
==1842== LL miss rate:          0.0% (        0.0%     +        0.0%  )
```

Figure 1

```
==2345==
==2345== Branches:      139,777,324  (139,773,671 cond +      3,653 ind)
==2345== Mispredicts:     1,072,599  (  1,072,250 cond +        349 ind)
==2345== Mispred rate:          0.7% (        0.7%     +        9.5%   )
```

Figure 2

That's fairly straightforward, the number of branches, branches mispredicted (conditional and indirect branches). Examples of C or C++ code that produces indirect branch machine code are calls through pointers to functions and virtual function calls. Conditional branches are generated for **if** statements and the conditional ternary operator.

```
desc: I1 cache:         65536 B, 64 B, 2-way associative
desc: D1 cache:         65536 B, 64 B, 2-way associative
desc: LL cache:         1048576 B, 64 B, 16-way associative
cmd: ./badprime
events: Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw Bc Bcm Bi Bim
fl=/build/buildd/eglibc-2.15/csu/../sysdeps/generic/dl-hash.h
fn=_init
44 1 0 0 0 0 0 0 0 0 0 0 0 0
45 1 0 0 0 0 0 0 0 0 0 0 0 0
46 13 0 0 4 1 0 0 0 0 4 3 0 0
49 16 0 0 0 0 0 0 0 0 0 0 0 0
50 8 0 0 0 0 0 0 0 0 0 0 0 0
63 8 0 0 0 0 0 0 0 0 0 0 0 0
68 1 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 3

So far, nothing to get too excited about. In the directory where I ran cachegrind there is now a file `cachegrind.out.2345` (where 2345 was the PID when it was executing, as you can see in the branch prediction snippet above). You can control the name of the file that cachegrind generates by using the **`-cachegrind-out-file`** option. Here's a small extract. It's not meant for human consumption (Figure 3).

You can make this slightly more digestible by using **`cg_annotate`**. You just need to enter **`cg_annotate cachegrind.out.<pid>`** and the output will go to the terminal. Figure 4 is what I got for this example.

Still it isn't very easy to read. You can see the same sort of notation as we saw for the overall summary (I – instruction, D – data and B – branch). I've cut the full paths to the file:function part. You can filter and sort the output with the **`-threshold`** and **`-sort`** options. You can also generate annotated source either with **`-auto=yes`** or on a file by file basis by passing the (fully qualified) filename as an argument to **`cg_annotate`**. I won't show an example here as it is rather long. Basically it shows the same information as in the **`cg_annotate`** output, but on a line by line basis. Lastly for **`cg_annotate`**, if you are using it for code coverage metrics, then you can combine multiple runs using **`cg_merge`**.

As already mentioned, you can use Kcachegrind. This is a fairly standard GUI application for browsing the performance results. It isn't part of Valgrind, rather it is a component of KDE so you may need to install it separately. If you want to be able to see the funky graphics, you'll need to have GraphViz installed. Figure 5 is a screen shot showing, on the left, percent of total time per function in descending order. The top 3, as expected, are the awful **`isPrime`**, **`sqrt`** and **`main`** with 99.9% between them. On the top right I've selected the Source Code tab, and we can see three lines with percent of time next to them. In this small example, the other tabs aren't very interesting.

Most of the navigating can be done by clicking the list of functions in the left pane. You can search for functions and also filter by the following groups – none, ELF object, source

Another problem with big real-world programs is speed ... there is a big overhead in performing the measurement

```
--------------------------------------------------------------------------------
I1 cache:         65536 B, 64 B, 2-way associative
D1 cache:         65536 B, 64 B, 2-way associative
LL cache:         1048576 B, 64 B, 16-way associative
Command:          ./badprime
Data file:        cachegrind.out.2345
Events recorded:  Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw Bc Bcm Bi Bim
Events shown:     Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw Bc Bcm Bi Bim
Event sort order: Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw Bc Bcm Bi Bim
Thresholds:       0.1 100 100 100 100 100 100 100 100 100 100 100 100
Include dirs:
User annotated:
Auto-annotation:  off

--------------------------------------------------------------------------------
         Ir   I1mr  ILmr           Dr  D1mr  DLmr          Dw  D1mw  DLmw          Bc        Bcm     Bi Bim
--------------------------------------------------------------------------------
922,466,826 1,234 1,183 349,345,693 7,557 5,119 10,114,555 1,555 1,197 139,773,671 1,072,250 3,653 349
PROGRAM TOTALS

--------------------------------------------------------------------------------
         Ir I1mr ILmr           Dr  D1mr  DLmr         Dw  D1mw DLmw          Bc      Bcm     Bi Bim
file:function
--------------------------------------------------------------------------------
895,017,739    1    0 340,937,515     0     0 5,000,000     0    0 135,559,306 968,532     0   0
badprime.cpp:isPrime(int)
 16,000,000    0    0   5,000,000     0     0 4,000,000     0    0   2,000,000       8     0   0
std::sqrt<int>(int)
 10,078,513    2    1   3,078,503     0     0 1,000,003     0    0   2,000,001  94,081     0   0
badprime.cpp:main
```

Figure 4

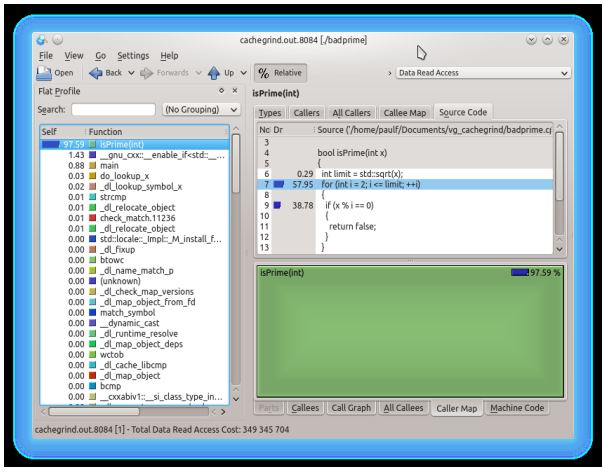


Figure 5

file, C++ class and function cycle. The two right panes have similar displays, roughly for displaying information about callers and callees (that is, functions called by the current function). Clicking on the **% Relative** toggle switches between showing absolute units (e.g., instruction fetches) and the percentage of the total. The drop down box on the top right allows you to display cache hit/miss rates, instruction fetches and cycle estimates.

Callgrind

For an example using callgrind, I downloaded and built a debug version of Icarus Verilog [Icarus] and compiled and simulated a Fibonacci calculator [Fibonacci] with small modifications to make the simulation last longer and exit on completion).

The commands that I used were:

```
iverilog -o fib fib*.v
```

for the compilation, and

```
valgrind --tool=callgrind
--log-file=callgrind.log vvp ./fib
```

to profile the simulation.

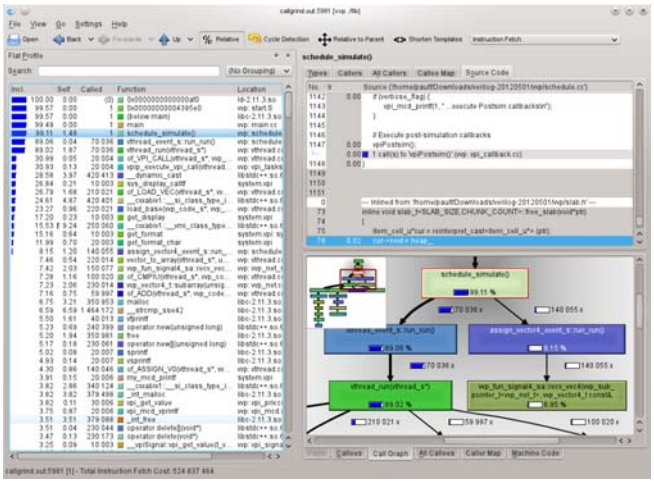


Figure 6

Finally

```
kcachegrind callgrind.out.5981
```

to view the results (see Figure 6).

This time, there isn't all of the cache hit/miss rate information, but instead there is far more on the interrelations between functions and time. You can navigate through the functions by double clicking in the call graph.

In the last snapshot, I show the funky callee map (which I expect will look better in the PDF version of the magazine in colour than in the black and white print magazine). The areas of the rectangles is proportional to the time spent in the function. Very pretty, but when it's as cluttered as this example, it's not much use. Both the map and the list are dynamic, and you can click in one and the area or list item will be highlighted in the other. You can also double click to jump to a different function. When you are navigating like this, you can use the back/forwards buttons on the toolbar to navigate in a similar fashion to a web browser, or navigate up the call stack (see Figure 7).

Callgrind has `callgrind_annotate` that is a text processor for its output files. This is the callgrind equivalent of `cg_annotate`, and since the output is very similar, I won't show it here.

Practical difficulties

One problem that I've experienced with callgrind was with differences between the floating point results with and without callgrind. In this case, I encountered numerical instability with callgrind that I didn't get when running the application natively. This is worse for 32bit x86, where by default code gets generated to use the venerable x87 floating point coprocessor. This uses 10byte (80bit) precision internally. Usually this means that if you have an expression that involves several operations on doubles, the operations will be done at 80bit precision and the result converted back to double precision (8byte, 64bit, required by the C and C++ standards). Valgrind performs the calculations at 64bit precision, so some of the intermediate precision is lost. Even with x64, which by default uses 64bit MMX/SSE for floating point calculations and so shouldn't have any internal truncation, I've still seen examples where the results are slightly different.

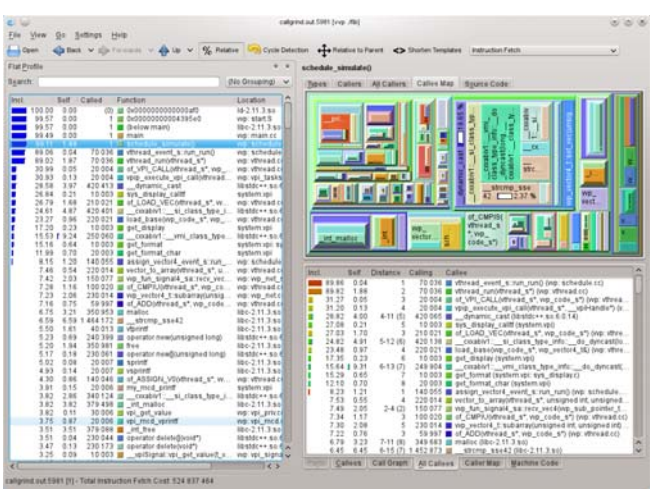


Figure 7

Another problem with big real-world programs is speed. Like all of the Valgrind tools, there is a big overhead in performing the measurement. You can mitigate this by controlling when callgrind performs data collection. The controlling can be done using command line options, using `callgrind_control --instr=<on|off>` or lastly by attaching gdb and using the monitor commands `instrumentation on|off`. You can statically control callgrind using Valgrind macros defined in `callgrind.h` (like those for memcheck that I described in my previous article on 'Advanced Memcheck' in *Overload 110*). You will probably need to do some profiling of the entire application to get an idea of where you want to concentrate your efforts.

In part 5 of this series, I'll be covering Massif, Valgrind's heap memory profiler. ■

References

[Fibonacci] http://www2.engr.arizona.edu/~slysecky/resources/verilog_tutorial.html

[Icarus] http://iverilog.icarus.com

Universal References in C++11

C++11 provides a new reference syntax, T&&. Scott Meyers explains that it doesn't always mean 'rvalue reference'.

Perhaps the most significant new feature in C++11 is rvalue references; they're the foundation on which move semantics and perfect forwarding are built. (If you're unfamiliar with the basics of rvalue references, move semantics, or perfect forwarding, you may wish to read Thomas Becker's overview before continuing. A full citation is provided in the 'Further information' section at the end of this article.)

Syntactically, rvalue references are declared like 'normal' references (now known as *lvalue references*), except you use two ampersands instead of one. This function takes a parameter of type rvalue-reference-to-**Widget**:

```
void f(Widget&& param);
```

Given that rvalue references are declared using **&&**, it seems reasonable to assume that the presence of **&&** in a type declaration indicates an rvalue reference. That is not the case:

```
Widget&& var1 = someWidget;
  // here, "&&" means rvalue reference

auto&& var2 = var1;
  // here, "&&" does not mean rvalue reference

template<typename T>
  void f(std::vector<T>&& param);
  // here, "&&" means rvalue reference

template<typename T>
  void f(T&& param);
  // here, "&&"does not mean rvalue reference
```

In this article, I describe the two meanings of **&&** in type declarations, explain how to tell them apart, and introduce new terminology that makes it possible to unambiguously communicate which meaning of **&&** is intended. Distinguishing the different meanings is important, because if you think 'rvalue reference' whenever you see **&&** in a type declaration, you'll misread a lot of C++11 code.

The essence of the issue is that **&&** in a type declaration sometimes means rvalue reference, but sometimes it means *either* rvalue reference *or* lvalue reference. As such, some occurrences of **&&** in source code may actually have the meaning of **&**, i.e., have the syntactic *appearance* of an rvalue reference (**&&**), but the *meaning* of an lvalue reference (**&**). References where this is possible are more flexible than either lvalue references or rvalue references. Rvalue references may bind only to rvalues, for example, and lvalue references, in addition to being able to bind to lvalues, may bind to rvalues only under restricted circumstances.[1] In contrast, references declared with **&&** that may be either lvalue references or rvalue references may bind to *anything*. Such unusually flexible references deserve their own name. I call them *universal references*.

The details of when **&&** indicates a universal reference (i.e., when **&&** in source code might actually mean **&**) are tricky, so I'm going to postpone coverage of the minutiae until later. For now, let's focus on the following rule of thumb, because that is what you need to remember during day-to-day programming:

> If a variable or parameter is declared to have type **T&&** for some **deduced type T**, that variable or parameter is a *universal reference*.

The requirement that type deduction be involved limits the situations where universal references can be found. In practice, almost all universal references are parameters to function templates. Because the type deduction rules for **auto**-declared variables are essentially the same as for templates, it's also possible to have **auto**-declared universal references. These are uncommon in production code, but I show some in this article, because they are less verbose in examples than templates. In the 'Nitty gritty details' section of this article, I explain that it's also possible for universal references to arise in conjunction with uses of **typedef** and **decltype**, but until we get down to the nitty gritty details, I'm going to proceed as if universal references pertained only to function template parameters and auto-declared variables.

The constraint that the form of a universal reference be **T&&** is more significant than it may appear, but I'll defer examination of that until a bit later. For now, please simply make a mental note of the requirement.

Like all references, universal references must be initialized, and it is a universal reference's initializer that determines whether it represents an lvalue reference or an rvalue reference:

- If the expression initializing a universal reference is an lvalue, the universal reference becomes an lvalue reference.
- If the expression initializing the universal reference is an rvalue, the universal reference becomes an rvalue reference.

This information is useful only if you are able to distinguish lvalues from rvalues. A precise definition for these terms is difficult to develop (the C++11 standard generally specifies whether an expression is an lvalue or an rvalue on a case-by-case basis), but in practice, the following suffices:

- If you can take the address of an expression, the expression is an lvalue.
- If the type of an expression is an lvalue reference (e.g., **T&** or **const T&**, etc.), that expression is an lvalue.
- Otherwise, the expression is an rvalue. Conceptually (and typically also in fact), rvalues correspond to temporary objects, such as those

Scott Meyers is one of the world's foremost authorities on C++ software development. Scott wrote the best-selling *Effective C++* series and is consulting editor for Addison Wesley's *Effective Software Development* series. He also conceived the boutique conferences 'The C++ Seminar' and 'C++ and beyond'. His website is aristeia.com

1. I discuss rvalues and their counterpart, lvalues, later in this article. The restriction on lvalue references binding to rvalues is that such binding is permitted only when the lvalue reference is declared as a reference-to-const, i.e. a const T&.

returned from functions or created through implicit type conversions. Most literal values (e.g., `10` and `5.3`) are also rvalues.

Consider again the following code from the beginning of this article:

```
Widget&& var1 = someWidget
auto&& var2 = var1;
```

You can take the address of `var1`, so `var1` is an lvalue. `var2`'s type declaration of `auto&&` makes it a universal reference, and because it's being initialized with `var1` (an lvalue), `var2` becomes an lvalue reference. A casual reading of the source code could lead you to believe that `var2` was an rvalue reference; the `&&` in its declaration certainly suggests that conclusion. But because it is a universal reference being initialized with an lvalue, `var2` becomes an lvalue reference. It's as if `var2` were declared like this:

```
Widget& var2 = var1;
```

As noted above, if an expression has type lvalue reference, it's an lvalue. Consider this example:

```
std::vector<int> v;
...
auto&& val = v[0];
// val becomes an lvalue reference (see below)
```

`val` is a universal reference, and it's being initialized with `v[0]`, i.e., with the result of a call to `std::vector<int>::operator[]`. That function returns an lvalue reference to an element of the vector.[2] Because all lvalue references are lvalues, and because this lvalue is used to initialize `val`, `val` becomes an lvalue reference, even though it's declared with what looks like an rvalue reference.

I remarked that universal references are most common as parameters in template functions. Consider again this template from the beginning of this article:

```
template<typename T>
  void f(T&& param);
  // "&&" might mean rvalue reference
```

Given this call to `f`,

```
f(10);  // 10 is an rvalue
```

`param` is initialized with the literal `10`, which, because you can't take its address, is an rvalue. That means that in the call to `f`, the universal reference `param` is initialized with an rvalue, so `param` becomes an rvalue reference – in particular, `int&&`.

On the other hand, if `f` is called like this,

```
int x = 10;
f(x);  // x is an lvalue
```

`param` is initialized with the variable `x`, which, because you can take its address, is an lvalue. That means that in this call to `f`, the universal reference `param` is initialized with an lvalue, and `param` therefore becomes an lvalue reference – `int&`, to be precise.

The comment next to the declaration of `f` should now be clear: whether param's type is an lvalue reference or an rvalue reference depends on what is passed when `f` is called. Sometimes `param` becomes an lvalue reference, and sometimes it becomes an rvalue reference. `param` really is a *universal reference*.

Remember that `&&` indicates a universal reference *only where type deduction takes place*. Where there's no type deduction, there's no universal reference. In such cases, `&&` in type declarations always means rvalue reference. Hence Listing 1.

There's nothing surprising about these examples. In each case, if you see `T&&` (where `T` is a template parameter), there's type deduction, so you're looking at a universal reference. And if you see `&&` after a particular type name (e.g., `Widget&&`), you're looking at an rvalue reference.

I stated that the *form* of the reference declaration must be `T&&` in order for the reference to be universal. That's an important caveat. Look again at this declaration from the beginning of this article:

2. I'm ignoring the possibility of bounds violations. They yield undefined behavior.

```
template<typename T>
void f(T&& param);
       // deduced parameter type ⇒ type deduction;
       // && ≡ universal reference
template<typename T>
class Widget {
  ...
  Widget(Widget&& rhs);
       // fully specified parameter type ⇒ no type
       // deduction; && ≡ rvalue reference
};
template<typename T1>
class Gadget {
  ...
  template <typename T2>
  Gadget(T2&& rhs);
       // deduced parameter type ⇒ type deduction;
  ...  // && ≡ universal reference
};
void f(Widget&& param);
       // fully specified parameter type ⇒ no type
       // deduction; && ≡ rvalue reference
```

Listing 1

```
template <class T,
          class Allocator = allocator<T> >
class vector {
public:
  ...
  void push_back(T&& x);
  // fully specified parameter type ⇒ no type
  // deduction; && ≡ rvalue reference
};
```

Listing 2

```
template<typename T>
void f(std::vector<T>&& param);
// "&&" means rvalue reference
```

Here, we have both type deduction and a `&&`-declared function parameter, but the form of the parameter declaration is not `T&&`, it's `std::vector<T>&&`. As a result, the parameter is a normal rvalue reference, not a universal reference. Universal references can only occur in the form `T&&`! Even the simple addition of a `const` qualifier is enough to disable the interpretation of `&&` as a universal reference:

```
template<typename T>
void f(const T&& param);
// "&&" means rvalue reference
```

Now, `T&&` is simply the required *form* for a universal reference. It doesn't mean you have to use the name `T` for your template parameter:

```
template<typename MyTemplateParamType>
void f(MyTemplateParamType&& param);
// "&&" means universal reference
```

Sometimes you can see `T&&` in a function template declaration where `T` is a template parameter, yet there's still no type deduction. Consider this `push_back` function in `std::vector`, shown in Listing 2.[3]

Here, `T` is a template parameter, and `push_back` takes a `T&&`, yet the parameter is not a universal reference! How can that be?

The answer becomes apparent if we look at how `push_back` would be declared outside the class. I'm going to pretend that `std::vector`'s `Allocator` parameter doesn't exist, because it's irrelevant to the discussion, and it just clutters up the code. With that in mind, here's the declaration for this version of `std::vector::push_back`:

3. `std::vector::push_back` is overloaded. The version shown is the only one that interests us in this article.

```
template <class T>
void vector<T>::push_back(T&& x);
```

`push_back` can't exist without the class `std::vector<T>` that contains it. But if we have a class `std::vector<T>`, we already know what `T` is, so there's no need to deduce it.

An example will help. If I write

```
Widget makeWidget();
// factory function for Widget
std::vector<Widget> vw;
...
Widget w;
vw.push_back(makeWidget());
// create Widget from factory, add it to vw
```

my use of `push_back` will cause the compiler to instantiate that function for the class `std::vector<Widget>`. The declaration for that `push_back` looks like this:

```
void std::vector<Widget>::push_back(Widget&& x);
```

See? Once we know that the class is `std::vector<Widget>`, the type of `push_back`'s parameter is fully determined: it's `Widget&&`. There's no role here for type deduction.

Contrast that with `std::vector`'s `emplace_back`, which is declared like Listing 3.

Don't let the fact that `emplace_back` takes a variable number of arguments (as indicated by the ellipses in the declarations for `Args` and `args`) distract you from the fact that a type for each of those arguments must be deduced. The function template parameter `Args` is independent of the class template parameter `T`, so even if we know that the class is, say, `std::vector<Widget>`, that doesn't tell us the type(s) taken by `emplace_back`. The out-of-class declaration for `emplace_back` for `std::vector<Widget>` makes that clear (I'm continuing to ignore the existence of the `Allocator` parameter):

```
template<class... Args>
void std::vector<Widget>::emplace_back
  (Args&&... args);
```

Clearly, knowing that the class is `std::vector<Widget>` doesn't eliminate the need for the compiler to deduce the type(s) passed to `emplace_back`. As a result, `std::vector::emplace_back`'s parameters are universal references, unlike the parameter to the version of `std::vector::push_back` we examined, which is an rvalue reference.

A final point is worth bearing in mind: the lvalueness or rvalueness of an expression is independent of its type. Consider the type `int`. There are lvalues of type `int` (e.g., variables declared to be `int`s), and there are rvalues of type `int` (e.g., literals like `10`). It's the same for user-defined types like `Widget`. A `Widget` object can be an lvalue (e.g., a `Widget` variable) or an rvalue (e.g., an object returned from a `Widget`-creating factory function). The type of an expression does not tell you whether it is an lvalue or an rvalue.

Because the lvalueness or rvalueness of an expression is independent of its type, it's possible to have *lvalues* whose type is *rvalue reference*, and it's also possible to have *rvalues* of the type *rvalue reference*:

```
Widget makeWidget();
// factory function for Widget

Widget&& var1 = makeWidget()
// var1 is an lvalue, but
// its type is rvalue reference (to Widget)

Widget var2 = static_cast<Widget&&>(var1);
// the cast expression yields an rvalue, but
// its type is rvalue reference (to Widget)
```

The conventional way to turn lvalues (such as `var1`) into rvalues is to use `std::move` on them, so `var2` could be defined like this:

```
Widget var2 = std::move(var1);
// equivalent to above
```

I initially showed the code with `static_cast` only to make explicit that the type of the expression was an rvalue reference (`Widget&&`).

Named variables and parameters of rvalue reference type are lvalues. (You can take their addresses.) Consider again the `Widget` and `Gadget` templates from earlier, shown in Listing 4.

In Widget's constructor, `rhs` is an rvalue reference, so we know it's bound to an rvalue (i.e., an rvalue was passed to it), but `rhs` itself is an lvalue, so we have to convert it back to an rvalue if we want to take advantage of the rvalueness of what it's bound to. Our motivation for this is generally to use it as the source of a move operation, and that's why the way to convert an lvalue to an rvalue is to use `std::move`. Similarly, `rhs` in `Gadget`'s constructor is a universal reference, so it might be bound to an lvalue or to an rvalue, but regardless of what it's bound to, `rhs` itself is an lvalue. If it's bound to an rvalue and we want to take advantage of the rvalueness of what it's bound to, we have to convert `rhs` back into an rvalue. If it's bound to an lvalue, of course, we don't want to treat it like an rvalue. This ambiguity regarding the lvalueness and rvalueness of what a universal reference is bound to is the motivation for `std::forward`: to take a universal reference lvalue and convert it into an rvalue only if the expression it's bound to is an rvalue. The name of the function (`forward`) is an acknowledgment that our desire to perform such a conversion is virtually always to preserve the calling argument's lvalueness or rvalueness when passing – *forwarding* – it to another function.

But `std::move` and `std::forward` are not the focus of this article. The fact that `&&` in type declarations may or may not declare an rvalue reference is. To avoid diluting that focus, I'll refer you to the references in the 'Further information' section for information on `std::move` and `std::forward`.

```
template <class T,
          class Allocator = allocator<T> >
class vector {
public:
  ...
  template <class... Args>
  void emplace_back(Args&&... args);
  // deduced parameter types ⇒ type deduction;
  ... // && ≡ universal references
};
```

Listing 3

```
template<typename T>
class Widget {
  ...
  Widget(Widget&& rhs);
  // rhs's type is rvalue reference, but rhs
  // itself is an lvalue
  ...
};
template<typename T1>
class Gadget {
  ...
  template <typename T2>
  Gadget(T2&& rhs);
  // rhs is a universal reference whose type will
  // eventually become an rvalue reference or an
  // lvalue reference, but rhs itself is an lvalue
  ...
};
```

Listing 4

Nitty gritty details

The true core of the issue is that some constructs in C++11 give rise to references to references, and references to references are not permitted in C++. If source code explicitly contains a reference to a reference, the code is invalid:

```
Widget w1;
...
Widget& & w2 = w1;  // error!
// No such thing as "reference to reference"
```

There are cases, however, where references to references arise as a result of type manipulations that take place during compilation, and in such cases, rejecting the code would be problematic. We know this from experience with the initial standard for C++, i.e., C++98/C++03.

During type deduction for a template parameter that is a universal reference, lvalues and rvalues of the same type are deduced to have slightly different types. In particular, lvalues of type **T** are deduced to be of type **T&** (i.e., lvalue reference to **T**), while rvalues of type **T** are deduced to be simply of type **T**. (Note that while lvalues are deduced to be lvalue references, rvalues are not deduced to be rvalue references!) Consider what happens when a template function taking a universal reference is invoked with an rvalue and with an lvalue:

```
template<typename T>
void f(T&& param);
...
int x;
...
f(10);  // invoke f on rvalue
f(x);   // invoke f on lvalue
```

In the call to **f** with the rvalue **10**, **T** is deduced to be **int**, and the instantiated **f** looks like this:

```
void f(int&& param);// f instantiated from rvalue
```

That's fine. In the call to **f** with the lvalue **x**, however, **T** is deduced to be **int&**, and **f**'s instantiation contains a reference to a reference:

```
void f(int& && param);
// initial instantiation of f with lvalue
```

Because of the reference-to-reference, this instantiated code is *prima facie* invalid, but the source code – **f(x)** – is completely reasonable. To avoid rejecting it, C++11 performs 'reference collapsing' when references to references arise in contexts such as template instantiation.

Because there are two kinds of references (lvalue references and rvalue references), there are four possible reference-reference combinations: lvalue reference to lvalue reference, lvalue reference to rvalue reference, rvalue reference to lvalue reference, and rvalue refer-ence to rvalue reference. There are only two reference-collapsing rules:

- An rvalue reference to an rvalue reference becomes ('collapses into') an rvalue reference.
- All other references to references (i.e., all combinations involving an lvalue reference) collapse into an lvalue reference.

Applying these rules to the instantiation of **f** on an lvalue yields the following valid code, which is how the compiler treats the call:

```
void f(int& param);  // instantiation of f with
// lvalue after reference collapsing
```

This demonstrates the precise mechanism by which a universal reference can, after type deduction and reference collapsing, become an lvalue reference. The truth is that a universal reference is really just an rvalue reference in a reference-collapsing context.

Things get subtler when deducing the type for a variable that is itself a reference. In that case, the reference part of the type is ignored. For example, given

```
int x;
...
int&& r1 = 10;  // r1's type is int&&
int& r2 = x;    // r2's type is int&
```

the type for both **r1** and **r2** is considered to be **int** in a call to the template **f**. This reference-stripping behavior is independent of the rule that, during type deduction for universal references, lvalues are deduced to be of type **T&** and rvalues of type **T**, so given these calls,

```
f(r1);
f(r2);
```

the deduced type for both **r1** and **r2** is **int&**. Why? First the reference parts of **r1**'s and **r2**'s types are stripped off (yielding **int** in both cases), then, because each is an lvalue, each is treated as **int&** during type deduction for the universal reference parameter in the call to **f**.

Reference collapsing occurs, as I've noted, in 'contexts such as template instantiation'. A second such context is the definition of **auto** variables. Type deduction for **auto** variables that are universal references is essentially identical to type deduction for function template parameters that are universal references, so lvalues of type **T** are deduced to have type **T&**, and rvalues of type **T** are deduced to have type **T**. Consider again this example from the beginning of this article:

```
Widget&& var1 = someWidget;
// var1 is of type Widget&& (no use of auto here)
auto&& var2 = var1;
// var2 is of type Widget& (see below)
```

var1 is of type **int&&**, but its reference-ness is ignored during type deduction in the initialization of **var2**; it's considered to be of type **Widget**. Because it's an lvalue being used to initialize a universal reference (**var2**), its deduced type is **Widget&**. Substituting **Widget&** for **auto** in the definition for **var2** yields the following invalid code:

```
Widget& && var2 = var1;
// note reference-to-reference
```

which, after reference collapsing, becomes:

```
Widget& var2 = var1;  // var2 is of type Widget&
```

A third reference-collapsing context is **typedef** formation and use. Given this class template,

```
template<typename T>
class Widget {
  typedef T& LvalueRefType;
  ...
};
```

and this use of the template,

```
Widget<int&> w;
```

the instantiated class would contain this (invalid) **typedef**:

```
typedef int& & LvalueRefType;
```

Reference-collapsing reduces it to this legitimate code:

```
typedef int& LvalueRefType;
```

If we then use this **typedef** in a context where references are applied to it, e.g.,

```
void f(Widget<int&>::LvalueRefType&& param);
```

the following invalid code is produced after expansion of the **typedef**,

```
void f(int& && param);
```

but reference-collapsing kicks in, so **f**'s ultimate declaration is this:

```
void f(int& param);
```

The final context in which reference-collapsing takes place is the use of **decltype**. As is the case with templates and **auto**, **decltype** performs type deduction on expressions that yield types that are either **T** or **T&**, and **decltype** then applies C++11's reference-collapsing rules. Alas, the type-deduction rules employed by **decltype** are not the same as those used during template or auto type deduction. The details are too arcane for coverage here (the 'Further information' section provides pointers to, er, further information), but a noteworthy difference is that **decltype**, given a named variable of non-reference type, deduces the type **T** (i.e., a non-reference type), while under the same conditions, templates and **auto** deduce the type **T&**. Another important difference is that **decltype**'s type

```
Widget w1, w2;
auto&& v1 = w1;
// v1 is an auto-based universal reference being
// initialized with an lvalue, so v1 becomes an
// lvalue reference referring to w1.
decltype(w1)&& v2 = w2;
// v2 is a decltype-based universal reference, and
// decltype(w1) is Widget, so v2 becomes an rvalue
// reference.
// w2 is an lvalue, and it's not legal to
// initialize an rvalue reference with an lvalue,
// so this code does not compile.
```

Listing 5

deduction depends only on the **decltype** expression; the type of the initializing expression (if any) is ignored. Ergo Listing 5.

Summary

In a type declaration, **&&** indicates either an rvalue reference or a *universal reference* – a reference that may resolve to either an lvalue reference or an rvalue reference. Universal references always have the form **T&&** for some deduced type **T**.

Reference collapsing is the mechanism that leads to universal references (which are really just rvalue references in situations where reference-collapsing takes place) sometimes resolving to lvalue references and sometimes to rvalue references. It occurs in specified contexts where references to references may arise during compilation. Those contexts are template type deduction, **auto** type deduction, **typedef** formation and use, and **decltype** expressions. ■

Acknowledgments

Draft versions of this article were reviewed by Cassio Neri, Michal Mocny, Howard Hinnant, Andrei Alexandrescu, Stephan T. Lavavej, Roger Orr, Chris Oldwood, Jonathan Wakely, and Anthony Williams. Their comments contributed to substantial improvements in the content of the article as well as in its presentation.

Further information

C++11, Wikipedia. http://en.wikipedia.org/wiki/C%2B%2B11

Overview of the New C++ (C++11), Scott Meyers, Artima Press, last updated January 2012. http://www.artima.com/shop/overview_of_the_new_cpp

'C++ Rvalue References Explained', Thomas Becker, last updated September 2011. http://thbecker.net/articles/rvalue_references/section_01.html

decltype, Wikipedia. http://en.wikipedia.org/wiki/Decltype

'A Note About decltype', Andrew Koenig, *Dr. Dobb's*, 27 July 2011. http://drdobbs.com/blogs/cpp/231002789

A DSEL for Addressing the Problems Posed by Parallel Architectures

Programming parallel algorithms correctly is hard. Jason McGuiness and Colin Egan demonstrate how a C++ DESEL can make it simpler.

Computers with multiple pipelines have become increasingly prevalent, hence a rise in the available parallelism to the programming community. For example the dual-core desktop workstations to multiple core, multiple processors within blade frames which may contain hundreds of pipelines in data centres, to state-of-the-art mainframes in the Top500 supercomputer list with thousands of cores and the potential arrival of next-generation cellular architectures that may have millions of cores. This surfeit of hardware parallelism has apparently yet to be tamed in the software architecture arena. Various attempts to meet this challenge have been made over the decades, taking such approaches as languages, compilers or libraries to enable programmers to enhance the parallelism within their various problem domains. Yet the common folklore in computer science has still been that it is hard to program parallel algorithms correctly.

This paper examines what language features would be required to add to an existing imperative language that would have little if no native support for implementing parallelism apart from a simple library that exposes the OS-level threading primitives. The goal of the authors has been to create a minimal and orthogonal DSEL that would add the capabilities of parallelism to that target language. Moreover the DSEL proposed will be demonstrated to have such useful guarantees as a correct, heuristically efficient schedule. In terms of correctness the DSEL provides guarantees that it can provide deadlock-free and race-condition free schedules. In terms of efficiency, the schedule produced will be shown to add no worse than a poly-logarithmic order to the algorithmic run-time of the schedule of the program on a CREW-PRAM (*Concurrent-Read*, *Exclusive-Write*, *Parallel Random-Access Machine* [Tvrkik99]) or EREW-PRAM (*Exclusive-Read EW-PRAM* [Tvrkik99]) computation model. Furthermore the DSEL described assists the user with regard to debugging the resultant parallel program. An implementation of the DSEL in C++ exists: further details may be found in [McGuiness09].

Related work

From a hardware perspective, the evolution of computer architectures has been heavily influenced by the von Neumann model. This has meant that with the relative increase in processor speed vs. memory speed, the introduction of memory hierarchies [Burger96] and out-of-order instruction scheduling has been highly successful. However, these extra levels increase the penalty associated with a miss in the memory subsystem, due to memory access times, limiting the ILP (Instruction-Level Parallelism). Also there may be an increase in design complexity and power consumption of the overall system. An approach to avoid this problem may be to fetch sets of instructions from different memory banks, i.e. introduce threads, which would allow an increase in ILP, in proportion to the number of executing threads.

From a software perspective, the challenge that has been presented to programmers by these parallel architectures has been the massive parallelism they expose. There has been much work done in the field of parallelizing software:

- Auto-parallelizing compilers: such as EARTH-C [Tang99]. Much of the work developing auto-parallelizing compilers has derived from the data-flow community [Snelling94].
- Language support: such as Erlang [Virding96], UPC [El-ghazawi03] or Intel's [Tian03] and Microsoft's C++ compilers based upon OpenMP.
- Library support: such as POSIX threads (*pthreads*) or Win32, MPI, OpenMP, Boost, Intel's TBB [Pheatt08], Cilk [Leiserson10] or various libraries targeting C++ [Giacaman08, Bischof05]. Intel's TBB has higher-level threading constructs, but it has not supplied parallel algorithms, nor has it provided any guarantees regarding its library. It also suffers from mixing code relating to generating the parallel schedule and the business logic, which would also make testing more complex.

These have all had varying levels of success, as discussed in part in [McGuiness06b], with regards to addressing the issues of programming effectively for such parallel architectures.

Motivation

The basic issues addressed by all of these approaches have been: correctness or optimization. So far it has appeared that the compiler and language based approaches have been the only approaches able to address both of those issues together. But the language-based approaches require that programmers would need to re-implement their programs in a potentially novel language, a change that has been very hard for business to adopt, severely limiting the use of these approaches.

Amongst the criticisms raised regarding the use of libraries [McGuiness06b, McGuiness06a] such as pthreads, Win32 or OpenMP have been:

- They have been too low-level so using them to write correct multi-threaded programs has been very hard; it suffers from composition problems. This problem may be summarized as: atomic access to an object would be contained within each object (using classic OOD), thus when composing multiple objects, multiple separate locks, from the different objects, have to be manipulated to guarantee correct access. If this were done correctly the usual outcome has been a serious reduction in scalability.
- A related issue has been that that the programmer often intimately entangles their thread-safety, thread scheduling and the business logic of their code. This means that each program would be

Jason McGuiness has given seminars internationally and to the BCS, IET and ACCU. He has worked on prototype supercomputers, which drove his interest in threading. Also Jason has been involved in the threading area of the C++11 standardization effort, and looks forward to contributing to the next C++ standard.

Colin Egan teaches on Advanced Computer Architecture, Computer Architecture and Principles of Networked Systems Administration and researches High Performance Processesor Design with particular reference to Energy Efficient Branch Prediction at Herfordshire University.

effectively a bespoke program, requiring re-testing of each program for threading issues as well as business logic issues.

- Also debugging such code has been found to be very hard. Debuggers for multi-threaded code have been an open area of research for some time.

Given that the language has to be immutable, a DSEL defined by a library that attempts to support the correctness and optimality of the language and compiler approaches and yet somehow overcomes the limitations of the usual library-based approaches would seem to be ideal. This DSEL will now be presented.

The DSEL to assist parallelism

We chose to address these issues by defining a carefully crafted DSEL, then examining its properties to demonstrate that the DSEL achieved the goals. The DSEL should have the following properties:

- The DSEL shall target what may be termed as *general purpose threading*, the authors define this to be scheduling in which the conditions or loop-bounds may not be computed at compile-time, nor could they be represented as monads, so could not be memoized[1]. In particular the DSEL shall support both data-flow and data parallel constructs.
- By being implemented in an existing language it would avoid the necessity of re-implementing the programs, a more progressive approach to adoption could be taken.
- It shall be a reasonably small DSEL, but be large enough to provide sufficient extensions to the host language that express parallel constructs in a manner that would be natural to a programmer using that language.
- It shall assist in debugging any use of a conforming implementation.
- It should provide guarantees regarding those banes of parallel programming: dead-locks and race-conditions.
- Moreover it should provide guarantees regarding the algorithmic complexity of any parallel schedule it would generate.

Initially a description of the grammar will be given, followed by a discussion of some of the properties of the DSEL. Finally some theoretical results derived from the grammar of the DSEL will be given.

Detailed grammar of the DSEL

The various types, production rules and operations that define the DSEL will be given in this section. The basic types will be defined first, then the operations upon those types will be defined. C++ has been chosen as the target language in which to implement the DSEL. This was due to the rich ability within C++ to extend the type system at compile-time: primarily using templates but also overloading various operators. Hence the presentation of the grammar relies on the grammar of C++, so it would assist the reader to have familiarity of that grammar, in particular Annex A of the ISO C++ Standard [ISO12]. Although C++11 has some support for threading, this had not been widely implemented at the time of writing, moreover the specification had not addressed the points of the DSEL in this paper.

Some clarifications:

- The subscript $_{opt}$ means that the keyword is optional.
- The subscript $_{def}$ means that the keyword is default and specifies the default value for the optional keyword.

Types

The primary types used within the DSEL are derived from the thread-pool type.

1. Thread pools can be composed with various subtypes that could be used to fundamentally affect the implementation and performance of any client software:

thread-pool-type:
`thread_pool` ***work-policy size-policy pool-adaptor***

- A thread pool would contain a collection of threads that may be more, less or the same as the number of processors on the target architecture. This allows for implementations to visualize the multiple cores available or make use of operating-system provided thread implementations. An implementation may choose to enforce a synchronization of all threads within the pool once an instance of that pool should be destroyed, to ensure that threads managed by the pool are appropriately destroyed and work in the process of mutation could be appropriately terminated.

work-policy: one of
`worker_threads_get_work one_thread_distributes`

- The library should implement the classic work-stealing or master-slave work sharing algorithms. Clearly the specific implementation of these could affect the internal queue containing unprocessed work within the **`thread_pool`**. For example a **`worker_threads_get_work`** queue might be implemented such that the addition of work would be independent to the removal of work.

size-policy: one of
`fixed_size tracks_to_max infinite`

- The *size-policy* when used in combination with the *threading-model* could be used to make considerable simplifications in the implementation of the *thread-pool-type* which could make it faster on certain architectures.
- **`tracks_to_max`** would implement some model of the cost of re-creating and maintaining threads. If thread were cheap to create & destroy with little overhead, then an **`infinite`** size might be a reasonable approximation, conversely threads with opposite characteristics might be better maintained in a **`fixed_size`** pool.

pool-adaptor:
joinability api-type threading-model priority-mode$_{opt}$ *comparator*$_{opt}$ *GSS(k)-batch-size*$_{opt}$

joinability: one of
`joinable nonjoinable`

- The *joinability* has been provided to allow for certain optimizations to be implementable. A *thread-pool-type* that is **`nonjoinable`** could have a number of simplifying details that would make it not only easier to implement but also faster in operation.

api-type: one of
`no_api MS_Win32 posix_pthreads IBM_cyclops`

- Both **`MS_Win32`** and **`posix_pthreads`** are examples of **`heavyweight_threading`** APIs in which threading at the OS-level would be made use of to implement the DSEL. **`IBM_cyclops`** would be an implementation of the DSEL using the lightweight_threading API implemented by IBM BlueGene/C Cyclops [Almasi03].

threading-model: one of
`sequential_mode heavyweight_threading lightweight_threading`

- This specifier provides a coarse representation of the various implementations of threadable construct in the multitude of architectures available. For example Pthreads would be considered to be **`heavyweight_threading`** whereas Cyclops would be **`lightweight_threading`**. Separation of the threading model versus the API allows for the possibility that there may be multiple threading APIs on the same platform,

1. A compile or run-time optimisation technique involving a space-time tradeoff. Re-computation of pure functions when provided with the same arguments may be avoided by caching the result; the result will be the same for each call with the same arguments, if the function has no side-effects.

which may have different properties, for example if there were to be a GPU available in a multi-core computer, there could be two different threading models within the same program.

- The **`sequential_mode`** has been provided to allow implementations to removal all threading aspects of all of the implementing library, which would hugely reduce the burden on the programmer regarding identifying bugs within their code. If all threading is removed, then all bugs that remain, in principle should reside in their user-code, which once determined to be bug-free, could then be trivially parallelized by modifying this single specifier and re-compiling. Then any further bugs introduced would be due to bugs within the parallel aspects of their code, or the library implementing this DSEL. If the user relies upon the library to provide threading, then there should be no further bugs in their code. We consider this feature of paramount importance, as it directly addresses the complex task of debugging parallel software, by separating the algorithm by which the parallelism should be implemented from the code implementing the mutations on the data.

priority-mode: one of

`normal_fifo`$_{\text{def}}$ **`prioritized_queue`**

- This is an optional parameter. The **`prioritized_queue`** would allow the user to specify whether specific instances of work to be mutated should be performed ahead of other instances of work, according to a user-specified *comparator*.

comparator:

`std::less`$_{def}$

- A unary function-type that specifies a strict weak-ordering on the elements within a the **`prioritized_queue`**.

GSS(k)-batch-size:

1_{def}

- A natural number specifying the batch-size to be used within the queue specified by the *priority-mode*. The default is 1, i.e. no batching would be performed. An implementation would be likely to use this for enabling GSS(k) scheduling [Kennedy02].

2. Adapted collections to assist in providing thread-safety and also specify the memory access model of the collection:

safe-colln:

`safe_colln` *collection-type lock-type*

- This adaptor wraps the *collection-type* and an instance of *lock-type* in one object, and provides a few thread-safe operations upon that collection, plus access to the underlying collection. This access might seem surprising, but this has been done because locking the operations on collections has been shown to not be composable, and cross-cuts both object-orientated and functional-decomposition designs. This could could be open to misuse, but otherwise excessive locking would have to be done in user code. This has not been an ideal design decision, but a simple one, with scope for future work. Note that this design choice within the DSEL does not invalidate the rest of the grammar, as this would just affect the overloads to the *data-parallel-algorithms*, described later.
- The adaptor also provides access to both read-lock and write-lock types, which may be the same, but allow the user to specify the intent of their operations more clearly.

lock-type: one of

`critical_section_lock_type read_write`
`read_decaying_write`

a. A **`critical_section_lock_type`** would be a single-reader, single-writer lock, a simulation of EREW semantics. The implementation of this type of lock could be more efficient on certain architectures.
b. A **`read_write`** lock is a multi-readers, single-write lock, a simulation of CREW semantics.
c. A **`read_decaying_write`** lock would be a specialization of a **`read_write`** lock that also implements atomic transformation of a write-lock into a read-lock.
d. The lock should be used to govern the operations on the collection, and not operations on the items contained within the collection.

- The *lock-type* parameter may be used to specify if EREW or CREW operations upon the collection are allowed. For example if EREW operations are only allowed, then overlapped dereferences of the **`execution_context`** resultant from *parallel-algorithms* operating upon the same instance of a `safe-colln` should be strictly ordered by an implementation to ensure EREW semantics are maintained. Alternatively if CREW semantics were specified then an implementation may allow read-operations upon the same instance of the *safe-colln* to occur in parallel, assuming they were not blocked by a write operation.

collection-type:

A standard collection such as an STL-style list or vector, etc.

3. The *thread-pool-type* defines further sub-types for convenience to the programmer:

`create_direct`:

This adaptor, parametrized by the type of work to be mutated, contains certain sub-types. The input data and the mutation operation combined are termed *the work to be mutated*, which would be a type of closure. If the mutation operation does not change the state of any data external to the closure, then this would be a type of monad. More specifically, this work to be mutated should also be a type of functor that either:

a. Provides a type **`result_type`** to access the result of the mutation, and specifies the mutation member-function,
b. or implements the function **`process(result_type &)`**, and the library may determine the actual type of **`result_type`**.

The sub-types are:

`joinable`: A method of transferring work to be mutated into an instance of *thread-pool-types*. If the work to be mutated were to be transferred using this modifier, then the return result of the transfer would be an **`execution_context`**, that may subsequently be used to obtain the result of the mutation. Note that this implies that the DSEL implements a form of data-flow operation.

`execution_context`: This is the type of future that a transfer returns. It is also a type of proxy to the **`result_type`** that the mutation returns. Access via this proxy implicitly causes the calling thread to wait until the mutation has been completed. This is the other component of the DSEL that implements the data-flow model. Various sub-types of **`execution_context`** exist specific to the **`result_types`** of the various operations that the DSEL supports. Note that the implementation of **`execution_context`** should specifically prohibit aliasing instances of these types, copying instances of these types and assigning instances of these types.

`nonjoinable`:

Another method of transferring work to be mutated into an instance of *thread-pool-types*. If the work to be mutated were to be transferred using this modifier, then the return result of the transfer would be nothing. The mutation within the pool would occur at some indeterminate time, the result of which would, for example, be detectable by any side effects of the mutation within the **`result_type`** of the work to be mutated.

`time_critical`:

This modifier ensures that when the work is mutated by a thread within an instance of *thread-pool-type* into which it has been

transferred, it will be executed at an implementation-defined higher kernel priority. Other similar modifiers exist in the DSEL for other kernel priorities. This example demonstrates that specifying other modifiers, that would be extensions to the DSEL, would be possible.

`cliques(natural_number n)`:

This modifier is used with *data-parallel-algorithms*. It causes the instance of *thread-pool-type* to allow the *data-parallel-algorithm* to operate with the number of threads shown, where p is the number of threads in the instance:

$$\left\lceil \frac{p}{n} \right\rceil$$

4. The DSEL specifies a number of other utility types such as **`shared_pointer`**, various exception types and exception-management adaptors amongst others. The details of these important, but ancillary types has been omitted for brevity.

Operators on the thread-pool-type

The various operations that are defined in the DSEL will now be given. These operations tie together the types and express the restrictions upon the generation of the control-flow graph that the DSEL may create.

1. The transfer work to be mutated into an instance of *thread-pool-type* is defined as follows:

   transfer-future:
   execution-context-result$_{\text{opt}}$
   thread-pool-type transfer-operation

   execution-context-result:
   `execution_context <<`

   - The token sequence **`<<`** is the transfer operation, and also used in the definition of the *transfer-modifier-operation*, amongst other places.
   - Note how an **`execution_context`** can only be created via a transfer of work to be mutated into the suitably defined **`thread_pool`**. It is an error to transfer work into a **`thread_pool`** that has been defined using the **`nonjoinable`** subtype. There is no way to create an **`execution_context`** with transferring work to be mutated, so every **`execution_context`** is guaranteed to eventually contain the result of a mutation.

   transfer-operation:
   transfer-modifier-operation$_{\text{opt}}$ *transfer-data-operation*

   transfer-modifier-operation:
   << *transfer-modifier*

   transfer-modifier: one of
   `time_critical joinable nonjoinable cliques`

   transfer-data-operation:
   << *transfer-data*

   transfer-data: one of
   work-to-be-mutated parallel-binary-operation
   data-parallel-algorithm

The details of the various *parallel-binary-operations* and *data-parallel-algorithms* will be given in the next section.

The data-parallel operations and algorithms

This section will describe the the various parallel algorithms defined within the DSEL.

1. The *parallel-binary-operations* are defined as follows:

   parallel-binary-operation: one of
   `binary_fun` *parallel-logical-operation*

   parallel-logical-operation: one of
   `logical_and logical_or`

   - It is likely that an implementation would not implement the usual short-circuiting of the operands, to allow them to transferred into the thread pool and executed in parallel.

2. The *data-parallel-algorithms* are defined as follows:

   data-parallel-algorithm: one of
   `accumulate copy count count_if fill fill_n find find_if for_each min_element max_element reverse transform`

   - The style and arguments of the *data-parallel-algorithms* is similar to those of the STL in the C++ ISO Standard. Specifically they all take a **`safe-colln`** as the arguments to specify the ranges and functors as necessary as specified within the STL. Note that these algorithms all use run-time computed bounds, otherwise it would be more optimal to use techniques similar to those used in HPF or described in [Kennedy02] to parallelize such operations. If the DSEL supports loop-carried dependencies in the functor argument is undefined.
   - If algorithms were to be implemented using techniques described in [Gibbons88] and [Casanova08], then the algorithms would be optimal with $O(\log(p))$ complexity in distributing the work to the thread pool. Given that there are no loop-carried dependencies, each thread may operate independently upon a sub-range within the safe-colln for an optimal algorithmic complexity shown below where n is the number of items to be computed and p is the number of threads, ignoring the operation time of the mutations.

     $$O\left(\frac{n}{p} - 1 + \log(p)\right)$$

3. The **`binary_funs`** are defined as follows:

   `binary_fun`:
   work-to-be-mutated work-to-be-mutated
   binary_functor

   - A *binary_functor* is just a functor that takes two arguments. The order of evaluation of the arguments is undefined. If the DSEL supports dependencies between the arguments is undefined. This would imply that the arguments should refrain from modifying any external state.

4. Similarly, the *logical_operations* are defined as follows:

   `logical_operation`:
   work-to-be-mutated work-to-be-mutated
   binary_functor

   - Note that no short-circuiting of the computation of the arguments occurs. The result of mutating the arguments must be boolean. If the DSEL supports dependencies between the arguments is undefined. This would imply that the arguments should refrain from modifying any external state.

Properties of the DSEL

In this section some results will be presented that derive from the definitions, the first of which will demonstrate that the CFG (*Control Flow Graph*) would be a tree from which the other useful results will directly derive.

Theorem 1

Using the DSEL described above, the parallel control-flow graph of any program that may use a conforming implementation of the DSEL must be an acyclic directed graph, and comprised of at least one singly-rooted tree, but may contain multiple singly-rooted, independent, trees.

Proof: From the definitions of the DSEL, the transfer of **`work to be mutated`** into the **`thread_pool`** may be done only once according to the definition of *transfer-future* the result of which returns a single **`execution_context`** according to the definition of *execution-context-result* which has been the only defined way to create **`execution_context`**s. This implies that from a node in the CFG, each transfer to the *thread-pool-type* represents a single forward-edge

connecting the **execution_context** with the child-node that contains the mutation. The back-edge from the mutation to the parent-node is the edge connecting the result of the mutation with the dereference of the **execution_context**. The **execution_context** and the dereference occur in the same node, because **execution_contexts** cannot be passed between nodes, by definition. In summary: the parent-node has an edge from the **execution_context** it contains to the mutation and a back-edge to the dereference in that parent-node. Each node may perform none, one or more transfers resulting in none, one or more child-nodes. A node with no children is a leaf-node, containing only a mutation. Now back-edges to multiple parent nodes cannot be created, according to the definition of **execution_context**, because **execution_context**s cannot be aliased nor copied between nodes. So the only edges in this sub-graph are the forward and back edges from parent to children. Therefore the sub-graph is not only acyclic, but a tree. Due to the definitions of *transfer-future* and *execution-context-result*, the only way to generate mutations is via the above technique. Therefore each child-node either returns via the back edge immediately or generates a further sub-tree attaching to the larger tree that contains it's parent. Now if the entry-point of the program is the single thread that runs **main()**, i.e. the single root, this can only generate a tree, and each node in the tree can only return or generate a tree, the whole CFG must be a tree. If there were more entry-points, each one can only generate a tree per entry-point, as the **execution_context**s cannot be aliased nor copied between nodes, by definition.

According to the above theorem, one may appreciate that a conforming implementation of the DSEL would implement data-flow in software.

Theorem 2

If the user refrains from using any other threading-related items or atomic objects other than those defined in the DSEL above then they can be guaranteed to have a schedule free of race-conditions.

Proof: A race-condition is when two threads attempt to access the same data at the same time. A race-condition in the CFG would be represented by a child node with two parent nodes, with forward-edges connecting the parents to the child. Note that the CFG must an acyclic tree according to theorem 1, then this sub-graph cannot be represented in a tree, so the schedule must be race-condition free.

Theorem 3

If the user refrains from using any other threading-related items or atomic objects other than those defined in the DSEL above and that the work they wish to mutate may not be aliased by any other object, then the user can be guaranteed to have a schedule free of deadlocks.

Proof: A deadlock may be defined as: when threads A and B wait on atomic-objects C and D, such that A locks C, waits upon D to unlock C and B locks D, waits upon C to unlock D. In terms of the DSEL, this implies that **execution_contexts** C and D are shared between two threads. i.e. that an **execution_context** has been passed from a node A to a sibling node B, and vice-versa occurs to **execution_context** B. But aliasing **execution_context**s has been explicitly forbidden in the DSEL by definition 3.

Corollary 1: If the user refrains from using any other threading-related items or atomic objects other than those defined in the DSEL above and that the work they wish to mutate may not be aliased by any other object, then the user can be guaranteed to have a schedule free of race-conditions and deadlocks.

Proof: It must be proven that the two theorems 2 and 3 are not mutually exclusive. Let us suppose that a CFG exists that satisfies 2 but not 3. Therefore there must be either an edge formed by aliasing an **execution_context** or a back-edge from the result of a mutation back to a dereference of an **execution_context**. The former has been explicitly forbidden in the DSEL by definition of the **execution_context**, 3, the latter forbidden by the definition of *transfer-future*, 1. Both are a contradiction, therefore such a CFG cannot exist. Therefore any conforming CFG must satisfy both theorems 2 and 3.

```
struct res_t {
  int i;
};
struct work_type {
  void process(res_t &) {}
};
typedef ppd::thread_pool<
  pool_traits::worker_threads_get_work,
  pool_traits::fixed_size,
  pool_adaptor<
    generic_traits::joinable,platform_api,
    heavyweight_threading
  >
> pool_type;
typedef pool_type::create_direct creator_t;
typedef creator_t::execution_context
execution_context;
typedef creator_t::joinable joinable;
pool_type pool(2);
execution_context context
   (pool<<joinable()<<work_type());
context ->i;
```

Listing 1

Theorem 4

If the user refrains from using any other threading-related items or atomic objects other than those defined in the DSEL above then the schedule of work to be mutated by a conforming implementation of the DSEL would be executed in time taking at least an algorithmic complexity of O(log(p)) and at most $O(n)$ in units of time to mutate the work where n is the number of work items to be mutated on p processors. The algorithmic order of the minimal time would be poly-logarithmic, so within NC, therefore at least optimal.

Proof: Given that the schedule must be a tree according to theorem 1 with at most n leaf-nodes, and that each node takes at most the number of computations shown below according to the definition of the *parallel-algorithms*.

$$O\left(\frac{n}{p}-1+\log(p)\right)$$

Also it has been proven in [Gibbons88] that to distribute n items of work onto p processors may be performed with an algorithmic complexity of $O(\log(n))$. The fastest computation time would be if the schedule were a balanced tree, where the computation time would be the depth of the tree, i.e. $O(\log(n))$ in the same units. If the n items of work were to be greater than the p processors, then $O(\log(p)) \leq O(\log(n))$, so the computation time would be slower than $O(\log(p))$. The slowest computation time would be if the tree were a chain, i.e. $O(n)$ time. In those cases this implies that a conforming implementation should add at most a constant order to the execution time of the schedule.

Some example usage

These are two toy examples, based upon an implementation in [McGuiness09], of how the above DSEL might appear. Listing 1 is a data-flow example showing how the DSEL could be used to mutate some work on a thread within the thread pool, effectively demonstrating how the future would be waited upon. Note how the **execution_context** has been created via the transfer of work into the **thread_pool**.

The **typedef**s in this example implementation of the grammar are complex, but the **typedef** for the *thread-pool-type* would only be needed once and, reasonably, could be held in a configuration trait in header file.

Listing 2 shows how a data-parallel version of the C++ **accumulate** algorithm might appear.

All of the parameters have been specified in the *thread-pool-type* to demonstrate the appearance of the **typedef**. Note that the example illustrates a map-reduce operation, an implementation might:

1. take sub-ranges within the *safe-colln*,

2. which would be distributed across the threads within the `thread_pool`,
3. the mutations upon each element within each sub-range would be performed sequentially, their results combined via the accumulator functor, without locking any other thread's operation,
4. These sub-results would be combined with the final accumulation, in this the implementation providing suitable locking to avoid any race-condition,
5. The total result would be made available via the `execution_context`.

Moreover the size of the input collection should be sufficiently large or the time taken to execute the operation of the accumulator so long, so that the cost of the above operations would be reasonably amortized.

Conclusions

The goals of the paper has been achieved: a DSEL has been formulated:

- that may be used to expresses general-purpose parallelism within a language,
- ensures that there are no deadlocks and race conditions within the program if the programmer restricts themselves to using the constructs of the DSEL,
- and does not preclude implementing optimal schedules on a CREW-PRAM or EREW-PRAM computation model.

Intuition suggests that this result should have come as no surprise considering the work done relating to auto-parallelizing compilers, which work within the AST and CFGs of the parsed program [Tang99].

It is interesting to note that the results presented here would be applicable to all programming languages, compiled or interpreted, and that one need not be forced to re-implement a compiler. Moreover the DSEL has been designed to directly address the issue of debugging any such parallel program, directly addressing this problematic issue. Further advantages of this DSEL are that programmers would not need to learn an entirely new programming language, nor would they have to change to a novel compiler implementing the target language, which may not be available, or if it were might be impossible to use for more prosaic business reasons.

Future work

There are a number of avenues that arise which could be investigated, for example a conforming implementation of the DSEL could be presented, for example [McGuiness09]. The properties of such an implementation could then be investigated by reimplementing a benchmark suite, such as SPEC2006 [Reilly06], and comparing and contrasting the performance of that implementation versus the literature. The definition of *safe-colln* has not been an optimal design decision a better approach would have been to define ranges that support locking upon the underlying collection. Extending the DSEL may be required to admit memoization could be investigated, such that a conforming implementation might implement not only inter but intra-procedural analysis. ■

```
typedef ppd::thread_pool<
  pool_traits::worker_threads_get_work,
  pool_traits::fixed_size,
  pool_adaptor<
     generic_traits::joinable,platform_api,
     heavyweight_threading,
     pool_traits::normal_fifo,std::less,1
  >
> pool_type;
typedef ppd::safe_colln<
    vector,<int >,
    lock_traits::critical_section_lock_type
> vtr_colln_t;
typedef pool_type::accumulate_t<
    vtr_colln_t
>::execution_context execution_context;
vtr_colln_t v;
v.push_back(1); v.push_back(2);
execution_context context(
  pool<<joinable()
    <<pool.accumulate(
      v,1,std::plus<vtr_coln_t::value_type>()
    )
);
assert(*context==4);
```

Listing 2

References

[Almasi03] George Almasi, Calin Cascaval, Jose G. Castanos, Monty Denneau, Derek Lieber, Jose E. Moreira, Jr. Henry S. Warren: 'Dissecting Cyclops: a detailed analysis of a multithreaded architecture', *SIGARCH Comput. Archit. News*, pp. 26–38, 2003.

[Bischof05] Holger Bischof, Sergei Gorlatch, Roman Leshchinskiy, Jens Müller: 'Data Parallelism in C++ Template Programs: a Barnes-hut Case Study', *Parallel Processing Letters*, pp. 257–272, 2005.

[Burger96] Doug Burger, James R. Goodman, Alain Kagi: 'Memory Bandwidth Limitations of Future Microprocessors'. In ICSA (1996) pp. 78–89, http://citeseer.ist.psu.edu/burger96memory.html

[Casanova08] H. Casanova, A. Legrand, Y. Robert: *Parallel Algorithms*. Chapman & Hall/CRC Press, 2008.

[El-ghazawi03] Tarek A. El-ghazawi, William W. Carlson, Jesse M. Draper: 'UPC language specifications v1.1.1', 2003.

[Giacaman08] Nasser Giacaman, Oliver Sinnen: 'Parallel iterator for parallelising object oriented applications'. In SEPADS'08: *Proceedings of the 7th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems*, pp. 44–49, 2008.

[Gibbons88] Alan Gibbons, Wojciech Rytter: *Efficient parallel algorithms*. Cambridge University Press, 1988.

[ISO12] ISO: *ISO/IEC 14882:2011 Information technology – Programming languages – C++*. International Organization for Standardization, 2012. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372

[Kennedy02] Ken Kennedy, John R. Allen: *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[Leiserson10] Charles E. Leiserson: 'The Cilk++ concurrency platform', *J. Supercomput.*, pp. 244–257, 2010. http://dx.doi.org/10.1007/s11227-010-0405-3.

[McGuiness06a] Jason M. McGuiness, Colin Egan, Bruce Christianson, Guang Gao: 'The Challenges of Efficient Code-Generation for Massively Parallel Architectures'. In *Asia-Pacific Computer Systems Architecture Conference* (2006), pp. 416–422.

[McGuiness06b] Jason M. McGuiness: 'Automatic Code-Generation Techniques for Micro-Threaded RISC Architectures', Masters' thesis, University of Herfordshire. 2006.

[McGuiness09] Jason M. McGuiness: 'libjmmcg – implementing PPD'. 2009. http://libjmmcg.sourceforge.net

[Pheatt08] Chuck Pheatt: 'Intel® threading building blocks', *J. Comput. Small Coll. 23*, 4 pp. 298–298, 2008.

[Reilly06] Jeff Reilly: 'Evolve or Die: Making SPEC's CPU Suite Relevant Today and Tomorrow'. In *IISWC*, pp. 119, 2006.

[Snelling94] David F. Snelling, Gregory K. Egan: 'A Comparative Study of Data-Flow Architectures', 1994. http://citeseer.ist.psu.edu/snelling94comparative.html

[Tang99] X. Tang: *Compiling for Multithreaded Architectures*. PhD thesis, 1999. http://citeseer.ist.psu.edu/tang99compiling.html.

[Tian03] Xinmin Tian, Yen-Kuang Chen, Milind Girkar, Steven Ge, Rainer Lienhart, Sanjiv Shah: 'Exploring the Use of Hyper-Threading Technology for Multimedia Applications with Intel® OpenMP* Compiler'. In *IPDPS*, p. 36, 2003.

[Tvrkik99] Pavel Tvrdik: 'Topics in parallel computing – PRAM models'. 1999. http://pages.cs.wisc.edu/~tvrdik/2/html/Section2.html

[Virding96] Robert Virding, Claes Wikström, Mike Williams: *Concurrent programming in ERLANG (2nd ed.)* (Armstrong, Joe, ed.). Prentice Hall International (UK) Ltd., 1996.

Keep It Simple, Singleton!

Naïve assumptions sound like a bad idea. Sergey Ignatchenko discusses how to spot good assumptions in the face of changing requirements.

Disclaimer: as usual, the opinions within this article are those of 'No Bugs' Bunny, and do not necessarily coincide with the opinions of the translator or the *Overload* editor. Please also keep in mind that translation difficulties from Lapine (like those described in [Loganberry2004]) might have prevented providing an exact translation. In addition, both the translator and *Overload* expressly disclaim all responsibility from any action or inaction resulting from reading this article.

There is a common wisdom in the programming world which says that the word 'naïve', when applied to anything programming-related, is essentially an insult (often aimed at the code, but usually also towards the author who has written such naïve code); as one of the authors has put it: 'I cannot remember a time when a working insightful function has been worse then the naïve equivalent' [Meyer2012]. However, as is usual with common wisdom, it is not as universal as it seems to be. There is at least one 'naïve' thing in the programming world which not only shouldn't be avoided, but which should be adhered to, especially in the agile programming world.

Naïve vs NotSoNaïve

Where there's a will, there's a way
~ proverb

Let us consider a simple example. Let us consider a hypothetical programming team which needs to write a plane simulator; it is stated in the specification that the plane to be simulated is a Cessna 162. So our team has started the design process, and the original version of the design (let's name it Naïve Design) has placed the **`Engine`** as a member of the class **`Plane`**:

```
// naive.h
class Plane
{
  Engine engine;
};
```

But then one of the younger team members Y (having noticed that a Cessna 162 is a single-engine aircraft, and being in the middle of reading the book *Design Patterns* [GoF1994]), said: 'Hey, why are we not using the singleton pattern here?' As he referred to the GoF book, which is considered a very reputable authority, the team has agreed to make both **`Engine`** and **`Plane`** singletons (see Listing 1).

The team (obviously, starting with team member Y) has been very excited about this change, saying that it not only looks much more professional and uses 'cool' newer technologies (as opposed to the `naive.h` approach, which was described as 'so eighties'), but also arguing that this approach reduces coupling. Although this is a good thing per se (while we won't comment here whether it really reduces coupling or not, this argument can easily be made and accepted in such context; if there is a will to use a certain pattern – there is a way).

```
// notsonaive.h
class Plane
{
  public:
    static Plane* Instance();

  private:
    Plane();
    static Plane* m_pInstance;
    //...
};

class Engine
{
  public:
    static Engine* Instance();

  private:
    Engine();
    static Engine* m_pInstance;
    //...
};
```

Listing 1

Incoming!

Welcome changing requirements, even late in development.
~ Agile Manifesto

Working hard, our hypothetical team has managed to write the Cessna 162 simulation software, and the simulation project turned out to be commercially successful. One momentous day, the manager addressed the team in very excited tone, saying, 'We have just secured a contract from the Big Company to extend our simulator to cover the whole line of Cessna aircraft'. It was certainly a good thing from a business perspective, with only one caveat: the whole line of Cessna aircraft includes the Reims-Cessna F406 Caravan II, which is a twin-engine aircraft, and support for twin-engine planes was a strict requirement by Big Company.

Obviously, the singleton **`Engine`** didn't allow for twin-engine planes, and unfortunately, dependencies on it being singleton were extensive and spread all over the code. It was at this point that the flight simulator program had to be redesigned almost from scratch, which delayed the software release by more than half a year, which in turn caused this second

'No Bugs' Bunny Translated from Lapine by Sergey Ignatchenko using the classic dictionary collated by Richard Adams.

Sergey Ignatchenko has 12+ years of industry experience, and recently has started an uphill battle against common wisdoms in programming and project management. He can be contacted at si@bluewhalesoftware.com

product to be late to market, and the project was closed pretty soon after the launch.

Moral of the story

The perfecting of one's self is the fundamental base of all progress and all moral development.
~ Confucius

The moral of this story was never intended as 'never use singletons' (though in our experience, singletons tend to be overused more than any other pattern), and certainly was not to imply that 'design patterns in general are bad'. This story was intended to illustrate two main points:

1. that to survive in the agile world, where changes of business task are frequent by definition, changes need to be anticipated (this has been explicitly said in the Agile Manifesto [AgileManifesto], so there is nothing new about it)

and

2. that 'naïve' mapping of the business task into software (such as classes and functions) tends to survive changes to business tasks significantly better than any artificial mappings.

Assumption is the mother of...

As soon as you have made a thought, laugh at it.
~ Lao Tzu

Let us analyze the situation in a bit more detail. What we're discussing here is essentially a mapping between two domains: the first one is a domain of business tasks, and the second one is a domain of programming solutions. If we take a look at NotSoNaïve mapping compared to Naïve mapping, it is easy to notice that a NotSoNaïve design essentially introduces two major assumptions into the picture: the first assumption is that there can be only one plane within the simulator, and the second one is that there can be only one engine within the simulator. It is the second assumption which was proven to be fatal in the case described above, however, even the first assumption could become bad enough (for example, if Big Company wants a simulator with more than one aircraft).

As usual, it was an invalid assumption which brought the design down. However, designing without assumptions is not usually feasible, so there is a question – how do we distinguish good assumptions from bad ones? Our hypothetical case study has suggested the answer to this question is as follows: only assumptions which are 'natural' from the point of view of the business task in hand, are 'good' ones. Obviously, this is not a strict definition, and answers may be different depending on who you ask, but still it is better than nothing.

One example of a 'reasonably good' assumption for our example of a flight simulator: while assuming that there is only one engine, one plane, and even one airport might be too restrictive, assuming that there is only one object Earth, is probably a reasonably good one (and therefore singleton object Earth is probably not a bad idea). It should be re-iterated, however, that even this 'reasonably good' assumption can fail, if, for example, there is a company which may want to use such a simulator in a sci-fi game covering multiple planets. Where exactly to draw the line (assuming that there is only one engine, or only one plane, or only one airport, or only one planet – or there are no 'only one' objects at all) – is a judgment call of the design team. What is important though, is that parameters to consider for this judgment call SHOULD include things like convenience and speed of development, and SHOULD NOT include things like 'using a cool technology' or (arguably even worse) 'where could we use this pattern?' logic.

Naïve mappings and SQL

The key, the whole key, and nothing but the key, so help me Codd
~ unknown

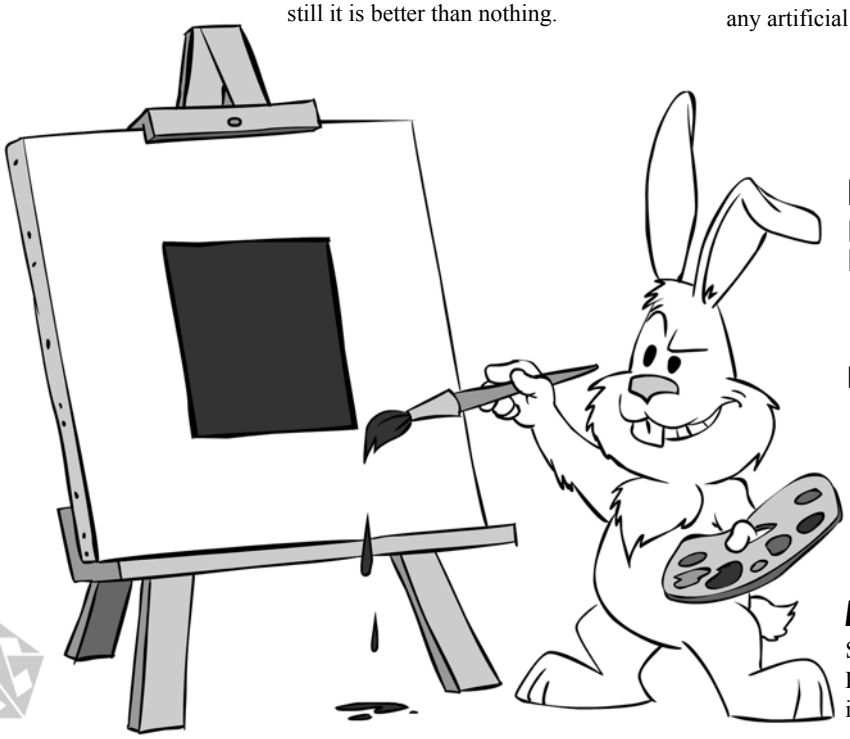
In the relational database world, data design usually starts with direct one-to-one mapping between business domain and programming domain, and 'non-naïve mappings' tend to be less of a problem. Still, even in the SQL world, it is possible to make some crazy decisions based on 'cool' technologies. One such scenario might happen if at the point of system design our team member Y is in the middle of reading a book on data warehousing; if he is persistent enough and there is nobody in the team who is bold enough to point out how inapplicable this model is to data processing, it may easily result in using 'cool' star architecture with fact tables and dimension tables for an operational database. Analysis of the consequences of such a brilliant decision is left as an exercise for the reader.

Summary

To summarize the above in the more formal way:

If we define 'naïve' mapping of business tasks into programs, as 'mapping without unnatural assumptions', and we define all other mappings as 'artificial mappings', and we define 'unnatural assumption' as 'assumption without sufficient justification from the nature of business task itself', we're claiming that:

> Naïve mappings of business tasks into software design tend to accommodate likely business task changes significantly better than any artificial mappings. ■



References

[AgileManifesto] http://agilemanifesto.org/

[GoF1994] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1994

[LoganBerry2004] David 'Loganberry', Frithaes! – an Introduction to Colloquial Lapine!, http://bitsnbobstones.watershipdown.org/lapine/overview.html

[Meyer2012] Shaun Meyer, Naive Programming, http://wordpress.morningside.edu/meyersh/2012/03/23/naive-programming/

Acknowledgement

Sergey Gordeev from Gordeev Animation Graphics, Prague, has provided the wonderful pictures that have illustrated this series of articles.

Software Developer Business Patterns

Patterns can be applied to business as well as software. Allan Kelly shows us how.

Paul Grenyer says: "I've read plenty of Allan Kelly's writings and they always make me think about things from a different point of view and I always learn something. I don't know if he sees everything differently to most people or just to me. I suspect it comes from Allan's experience across every stage of the product lifecycle."

"I read and enjoyed Allan's first book, *Changing Software Development* and looked forward to *Business Patterns for Software Developers*. It came out at a time shortly after I had left a startup company. It helped me understand a lot of what had happened and why. As with all good patterns books it helped me recognise and name patterns. Again, I learnt a lot and the patterns are proving useful as I am now shaping my own startup."

Here we go again, Apps. Even the word *application* is too big for the small pieces of software we download and install on our phones, tablets or other gadget. Apps are a wonderful reminder of the power of software. Indeed, software apps sit within a much bigger information, technology and communications ecology which is itself powered by software!

iOS Apps, Android Apps, Facebook Apps and now Apps for televisions. We've been here before, last time it was the web, and before that Windows, before that MSDOS and before that Apple IIs and TSR-80s.

Every seven years or so our industry comes up with a new paradigm which allows small pieces of software, sometimes written by one person, to advance the ecology itself. A few make millions while many have walk-on parts.

That is the way the software industry is: high growth, high failure, dynamic, loved of politicians and venture capitalists. The barriers to entry are shockingly low: as long as you can code, have an idea and can live on credit cards for a few months you can play too.

Yet the barriers to success are extremely high. Just because you can play doesn't mean you will succeed. Infant mortality is shocking. And we repeat the same mistakes over and over again.

Perhaps the biggest mistake is to believe that if we build it they will come. Those days are over, if they ever existed in the first place.

Successful software companies know about marketing, product strategy, roadmaps, customers and many other things. In fact, successful software companies have a lot in common. There are, just as in code, patterns of success, patterns of business.

Over the last eight years I have collected many of these patterns. Often identifying and writing one pattern leads to the discovery of several more. All patterns are available for download from my website and 36 were published as a book in early 2012. In the rest of this piece I'd like to discuss a few of the patterns and how they fit together.

Same customers, different product

Once your business has a relationship with your customers they will trust you, satisfied customers have a positive impression of your company and your people. Sales staff already have a relationship with the customers and

Same Customer, Same Product is normally found with consumable

Same customer	Consumables, e.g. repeated razor blades or printer ink. Sell to other groups within the same corporate.	Customer is the asset, find or develop new products to sell to the customer.
Different customer	Product is the asset view.	Diversification: a new product in a new market.
	Same product	**Different product**

Figure 1

the company is in a good position both to understand their needs and to supply them.

Modern consumers are faced with a bewildering number of brands and product variations. Choosing a trusted brand is one way to navigate the jungle.

Nor is it just individual consumers who return to existing suppliers and brand. Business customers find it easier to buy from suppliers they have worked with before; and the promise of a compatible product is a big plus.

So rather than continually seek out new customers for the product you have consider the customers – rather than your products – as the asset. Expand your product offering so you have more products to sell to your existing customers. Do this by focusing on the customer.

This approach has the additional benefit that by deepening your relationship with the customer not only will you sell more, the customer will get a better solution and the relationship will be strengthened further.

Some sales will build directly on the products you have already sold – sometimes called 'plus one' sales: more licences for the same software, upgrades to enhanced versions new versions of existing software, and add-on modules are all possible. The next sale could be a service: support contracts, training and installation are highly applicable to existing products.

In fact, when you think about it there are actually four possibilities for sales, as shown in Figure 1.

Different Customer, Same Product is probably the default position of most companies – 'We have a product, we need to find more customers!' Yet making sales can be very expensive in itself, customers might already be using a competitor product or new sales channels might be needed to reach new customer groups.

Allan Kelly has held just about every job in the software world. Today he provides training and coaching to teams and companies in the use of Agile and Lean technologies to develop better software with better processes. He is the author of Changing Software Development: Learning to become Agile, numerous journal articles and is currently working on a book of Business Strategy Patterns. Contact him at http://www.allankelly.net

As you decide which customer groups to serve, you will define your market position relative to your competitors

products – Gillette razor blades or HP ink jet cartridges but it could be software upgrades, I must have bought five or six copies of Microsoft Office in my life.

When products last like software ways need to be found to entice the same customer to buy more of the same product. If the customer is a large organization then it might be possible to move from one group within the company to another selling licences as you go. While the paying customer might be the same the end user will be different.

For home users it might be an upgrade cycle – new OS, new PC, new Office version. Although even this might be considered a Same Customer, Different Product model depending on how you view upgrades.

Diversification – Different Customer, Different Product – is inherently risky because it is, in effect, establishing a new business. Sometimes companies pull this off, like the way Caterpillar extended its brand into clothing. Perhaps more often this is one jump too many, look at the way many tobacco companies have tried to move into food only to retreat later.

Same Customers, Different Product view customers, not the product, as the asset. Rather than sell an existing product to new customers you sell existing customer a new product. When done well this can be highly effective at low risk. Think of Oracle building on the database customer base to sell enterprise software and middleware, or the IBM Rational developer toolset.

Nor does the different product need to be your own product. You could use the patterns Value Added Reseller or White Label to offer products to your existing customer base under your own name. Offering another product as a complement to your own can reduce costs and eliminate competitive tension – something described in the pattern Complementor Not Competitor.

Of course it is sometimes difficult from the outside to tell what a company's strategy actually is. Consider Dell's recent acquisition of Quest Software. This might be a diversification move – Different Customer, Different Product – or attempt to sell software to existing hardware customers, or possibly vice versa.

Homogenous and segmented customers

In the beginning many companies assume – by conscious decision or through naivety – that all customers are homogenous. Indeed, this approach has some advantages: it reduces time to market entry and therefore costs. By entering the market it puts the company in the best possible position to understand what customers want and what is possible.

A homogenous customers approach has worked well for the likes of Microsoft – with MSDOS – and even Google's early search product. If you consider all customers to be the same then one product is all you need to enter the market – thus keeping costs down and time short.

In the beginning companies may simply lack an understanding of potential customers. Sometimes the best way to gain knowledge about customers is simply to get into the market. However, sooner or later companies start segmenting their customer base so to serve customers better, and to extract more money.

Segmenting makes sense: customers are not homogenous. Different customers want different things, some value speed or ease of use, others value low price. Many a software development team has been lead astray by focusing on the needs of one customer too closely and producing a product which has little, or no, use to others.

The trick is to use Customer Understanding (another pattern) to segment customers into different groups and address the needs of each group separately. Segment groups are typically defined on discernable attributes and characteristics that allow differentiation of one group from another; another approach is to segment on the tasks to be performed. Either way, working with definable groups avoids generalisations that do not accurately describe any one group.

You may choose not to meet the needs of some groups if doing so would compromise the needs of another. When resources are limited is it better to target resources than spread them thin? Segmenting your customers will allow you to segment your market. As you decide which customer groups to serve, you will define your market position relative to your competitors. You are also deciding who you will not serve. In some cases you may need to extricate yourself from some existing group to pursue your new targets.

Segmentation can help avoid situations where more attention is paid to the customer who shouts loudest. Strategy is as much, or even more, about what you will not do, who you will not serve, as it is what you will do and who you will serve.

Those you decide to serve will benefit because they will get a product that more specifically fits their needs. A deliberate choice not to serve some groups allows you to serve others better. Sub-dividing your customer base will also help you spot opportunities to serve customers better.

But customer segmentation can go too far. While for customers the sheer choice of products on offer under the same brand can be overwhelming. Last year I needed to buy a new car GPS system, confronted with what seemed like several thousand products on Amazon I stuck with the brand I knew, Tom-Tom, and bought the cheapest product rather than spend time considering the merits of each product. (True, I momentarily considered a Garmin but was again overwhelmed by product choices.)

More dramatically contrast the approach of Nokia and Apple to the phone market. Nokia markets an amazing range of different phones. Conceivably there is a Nokia for every customer segment – indeed there might well be more than one. There comes a point were the costs of such a diversified product range become self-defeating.

Now consider Apple's approach to the same market: there is one. There was one iPhone to start with and although Apple continues to offer older versions, and some Simple Product Variations (another pattern) there is essentially still one iPhone.

Apple too have segmented the market but having done so they decided to only address the premium smart phone market. Even here segmentation is limited. A quick look at Nokia's website one day in June 2012 shows four different Lumia Smartphones, six other smart phones and at least a dozen simpler phones.

addressing a segmented customer base with a range of products targeted at each segment should be a more effective approach than homogenous customers

On the face of it addressing a segmented customer base with a range of products targeted at each segment should be a more effective approach than homogenous customers. But as the Apple v. Nokia story shows things aren't always that simple.

Sequences and competitive advantage

Patterns don't exist in isolation. If one company faces a problem and solves it using the same pattern as another it is quite likely to find the next problem is also quite similar and the corresponding solution is also similar.

Figure 2 shows one such pattern sequence. The start of this sequence is the desire to reach more customers. After segmenting the customer base the company engages Simple Product Variations – maybe a range of colours products al la the iMac or iPad covers – and Core Product Only. (While Core Product Only works in some domains it doesn't have great history in the IT world. Consider Microsoft Works, it has most of the Office functionality most users need but has never sold well. Even Linux seldom appears on its own, it is usually the core of a distribution or baked into a product like Android.).

Consequently the company has a Product Portfolio. In order to reach the different customer segments – and limit competition with itself – the company then engages a variety of distribution channels.

While one pattern may address one set of problems right here and now in resolving the question new issues arise. Thus patterns tend to lead to other patterns – a pattern of patterns usually know as a pattern sequence.

That different companies follow the same patterns and same sequences isn't a bad thing, in fact it validates the whole approach. For many companies the product, not the process or organization, is the competitive advantage. Why reinvent the process wheel when plenty of others have already shown how to play the game?

That said for some reinventing the wheel is the competitive advantage. The way the product or service is delivered, or the way the company works, could well be the things that make the company a winner. The actual product may be very similar to the competition but delivery very different. For these companies business patterns serve not as a template for what to do but rather a description of what not to do.

Finally

The software industry, indeed the wider technology industry, doesn't stand still. As the patterns were re-edited for the book it became clear that some updates were needed.

For example, Nokia was sited as an example in several patterns. In some cases the pattern and example still held. In others, the recent events at Nokia lead to new insights into the business of technology.

I believe the patterns themselves will stand the test of time but I expect some of the details and examples will change. I also expect that using this common languages for much of what our industry does will allow software developers and entrepreneurs to come up with new variations, new combinations and completely new approaches.

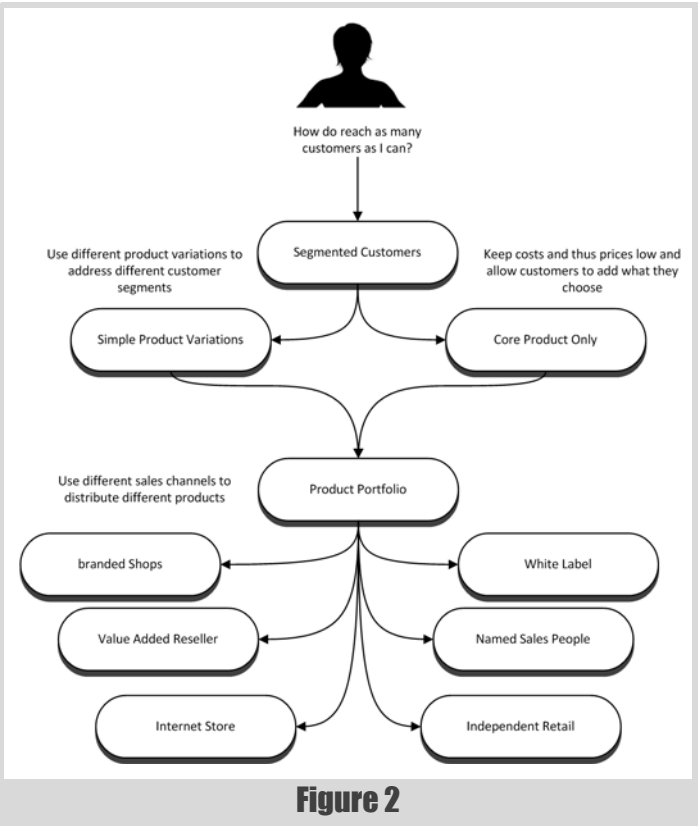


Figure 2

Complementor, Not Competitor

Strawberries do not compete with cream.

> Three years, a lot of activity, and a few billion dollars later, we still weren't [application software] leaders...
>
> However, one thing we were doing exceptionally well was irritating the heck out of the leading application providers – companies like SAP, PeopleSoft and JD Edwards. These companies were in a great position to generate a lot of business for us...
>
> What we said to them was 'We are going to leave this market to you; we are going to be your partners rather than your competitor'. (Gerstner, 2003)

Context

Within your product portfolio you sell two or more products that are complementary and usually sold together; it might not even make sense to buy them individually. Your competitors offer one or other of the products, and customers often mix-and-match. One of your products can hold its own against the competition, but the other cannot.

Problem

How do you arrange your product and services portfolio so that you maximise your profits and don't lose money on products?

Forces

You have two complementary products: let's call them 'Strawberries' and 'Cream'. Strawberries sells well and makes money, but Cream sells less well; it loses money, and even some of your Strawberry customers prefer to buy their Cream from another vendor.

You have traditionally sold customers a total solution of both Strawberries and Cream. But, while your Strawberries are very good, there is better Cream on offer elsewhere. Customers who buy Strawberries elsewhere are unlikely to buy your Cream too.

Developing both Strawberries and Cream has allowed for innovation in the past, but now it is competitors who sell only Strawberries or only Cream who are generating the innovations. Having both does not offer a lot more opportunities for innovation.

Traditionally your own Strawberries and Cream, and the innovation possible, has given you a strategic advantage. But the market, business strategy and core competencies have changed over time, and you no longer get the advantage you once did.

Selling both products together makes for a bigger sale with more revenue, but when costs are broken down, most of the profit comes from the Strawberries.

Both Strawberries and Cream are expensive to develop; while Strawberries makes money, Cream is losing money.

Solution

Concentrate your activities on the most profitable part of the solution; discontinue the less profitable parts and replace the missing parts with ones from other vendors. Rather than competing with everyone, seek to complement those who can help you sell more of your most profitable products.

Customers will want to buy Strawberries and Cream; now that you no longer compete with the Cream manufacturers, they can be a source of customers. You will need to form partnerships to align your goals and methods. Your staff will need to know partners' products and ensure partners' Cream works well with your Strawberries.

You need to prove to your new partners that you are committed to this strategy. Move fast and decisively to show that you are now a friend and not an enemy. Cream makers could still recommend one of the other Strawberry suppliers, so work together to produce the best Strawberries for their Cream.

Consequences

You no longer supply a total solution with your own products; instead, you sell your most profitable product and complement it with third-party products to offer a total solution.

Opportunities for innovation between Strawberries and Cream are more difficult to find and exploit; however, you can be more focused on innovation for Strawberries.

Discontinuing a loss-making product should immediately help your balance sheet. By partnering with others you can increase sales revenue for your profitable products – a classic win-win situation.

Sales of Strawberries alone may be smaller, but they will be more profitable; in addition, you are hoping to sell more Strawberries by working together with the Cream makers. You may be able to make revenue from consultancy services too.

You have saved the cost of developing an expensive product. Product development costs are lower; the cash released may be used elsewhere, say in new products or services.

However, you will need to ensure that your Strawberries are compatible with the Cream from various vendors. Ensuring compatibility can in itself be a timely and costly endeavour.

Customers are no longer locked into your products; they now choose your products because you have the best solution to their needs, not because they have no choice.

New partners may seek to lock you into their product; if you become dependent on one Cream maker, you will be in a weak position if they ask for special consideration and price cuts. So work with several Cream makers to give you the choice to walk away from a deal if a partner asks too much.

Variations

Products may be service products. You might stop offering your physical product while offering your services in support of your traditional competitors' products. Conversely, you may discontinue your own services offerings and encourage third-party suppliers to offer services to support your products.

Examples

Games console manufactures usually lose money on each console sold while making large profits on the games for the consoles. After selling over 6 million Dreamcast consoles and losing $500 million, Sega left the market in 2001 and chose to focus on producing software for Sony, Microsoft and Nintendo consoles.

Related patterns

Contrast with LIFETIME SERVICES FOR PRODUCTS, where the sale of one product – possibly at a loss – allows you to make money from a second.

IN BED WITH THE ENEMY (Weiss, 2007) also describes how companies can work with former and even current competitors.

A WHOLE PRODUCT strategy can lead companies to produce supporting products that in time end up losing money in their own right. As long as customers see a total solution to their problems, it is not important whether all components come from one supplier or many. ■

References

[Gerstner03]Gerstner, L. V. 2003. *Who says Elephants Can't Dance?*, London, HarperCollins

[Weiss07] 'In Bed with the Enemy' in Hvatum, L. & Schuemmer, T., eds. *EuroPLoP – Proceedings of the 8th European Conference on Patterns Languages of Programs*, 2007 Irsee, Germany. UVK Universitätsverlag Konstanz GmbH