A Declarative Language Approach to Device Configuration

Adrian Schüpbach Andrew Baumann [*] Timothy Roscoe Simon Peter

Systems Group, Department of Computer Science, ETH Zurich
www.systems.ethz.ch

Abstract

C remains the language of choice for hardware programming (device drivers, bus configuration, etc.): it is fast, allows low-level access, and is trusted by OS developers. However, the algorithms required to configure and reconfigure hardware devices and interconnects are becoming more complex and diverse, with the added burden of legacy support, "quirks", and hardware bugs to work around. Even programming PCI bridges in a modern PC is a surprisingly complex problem, and is getting worse as new functionality such as hotplug appears. Existing approaches use relatively simple algorithms, hard-coded in C and closely coupled with low-level register access code, generally leading to suboptimal configurations.

We investigate the merits and drawbacks of a new approach: separating hardware configuration *logic* (algorithms to determine configuration parameter values) from *mechanism* (programming device registers). The latter we keep in C, and the former we encode in a declarative programming language with constraint-satisfaction extensions. As a test case, we have implemented full PCI configuration, resource allocation, and interrupt assignment in the Barrelfish research operating system, using a concise expression of efficient algorithms in constraint logic programming. We show that the approach is tractable, and can successfully configure a wide range of PCs with competitive runtime cost. Moreover, it requires about half the code of the C-based approach in Linux while offering considerably more functionality. Additionally it easily accommodates adaptations such as hotplug, fixed regions, and "quirks".

Categories and Subject Descriptors D.1.6 [*Software*]: Programming Techniques—Logic Programming; D.4.9 [*Software*]: Operating Systems—Systems Programs and Utilities

General Terms Algorithms, Design, Languages

Keywords Constraint logic programming, Eclipse CLP, Hardware programming, PCI configuration

1. Introduction

Although many attempts have been made to improve on it, C remains the language of choice for writing code to program hardware, including device drivers, bus configuration, and interrupt routing. C is fast, provides low-level access to hardware registers, and is trusted by OS developers.

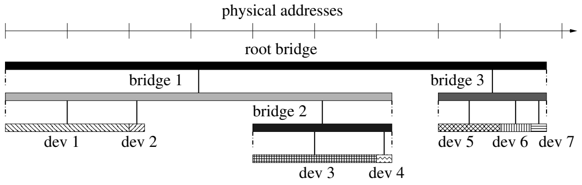


Figure 1. Example PCI tree with one root, three bridges, and 7 devices, showing the decoding of addresses from one physical memory space (e.g. non-prefetchable). Bridge base addresses are bounded by the union of the base and limit addresses of their children.

However, trends in hardware are making efficient and correct OS code for hardware access more difficult to write. Hardware platforms and system interconnects are becoming more complex and diverse, while at the same time it is increasingly important for overall performance to derive efficient configurations of devices, interrupts, and memory regions.

Configuring the hardware, I/O bridges and memory regions by interacting with platform firmware, is a surprisingly complex problem in a modern computer. The same is true for allocating and routing interrupts, handling device hotplug, etc., and it is getting worse as new functionality appears. Existing operating systems code uses relatively simple algorithms to achieve these goals. These algorithms are simple by the necessity of being hard-coded: they require low-level access to device registers to achieve their goals, and usually run early at system start-up within the OS kernel.

Figure 1 illustrates a simplified PCI-based device configuration, and the way that it is handled by typical operating systems. The OS code must allocate memory regions to each PCI device, and each PCI bridge in the bus hierarchy, in such a way that every device receives correctly-sized areas of memory in distinct regions (prefetchable, and non-prefetchable) of two different address spaces (I/O and memory). These areas must all be aligned to device-specific boundaries, may not overlap, and should fit into the total amount of physical address space available for such hardware in the system.

We describe the PCI configuration problem in detail in Section 2, but two factors make this allocation problem particularly hard. First, hotplugging means that devices can come and go in the hierarchy, which may entail reconfiguring entire subtrees, which is in turn disruptive to running device drivers. Second, there are numerous restrictions on device allocation: certain devices or bridges must be placed at a fixed address, others incorrectly decode addresses not assigned to them, and platform hardware components such as ACPI sometimes reserve regions of physical address space, which means that the address ranges must be allocated "around" these holes. As computer architectures become more complex, this list of problems is likely to grow, and to vary widely from one sys-

[*] Now at Microsoft Research.

ASPLOS'11, March 5–11, 2011, Newport Beach, California, USA.

tem to another. We fully expect to see analogous issues for future interconnects or platform functions.

Most existing OSes deal with this problem with simple algorithms in C such as sorting devices by address range size, modified with much special-case code. The result is complex and hard to debug, and (as we show in Section 5) can lead to unpredictable and inefficient allocation of space as devices are hotplugged. In some cases such as Linux on Intel platforms, the OS does not even try to solve the allocation problem, instead relying on the platform BIOS to provide an initial allocation, which is difficult to change.

Our aim is to find techniques for this general class of resource allocation that result in cleaner, smaller, more flexible code which still accommodates the various quirks, bugs, and legacy restrictions imposed by real-world hardware. Our goal is to make such OS code easier to write and evolve over time, and more reliable in the face of ever-more-complex hardware.

In this paper we investigate the costs and benefits of a radically different approach: separating configuration *logic*, such as the algorithms to determine which configuration parameter values should be employed, from the configuration *mechanism* (actually reading and writing device registers). The latter we keep in C as part of the kernel, but the former we encode in a logic programming language with constraint-solving extensions in the system knowledge base [26], running as an OS service.

Hardware-related code can be roughly divided into "data path" functionality (interrupt handlers, packet processing, descriptor management, etc.), and configuration management (PCI programming, ACPI initialization and interpretation, memory region and I/O space allocation, etc.). Both are critical to the performance and correct functioning of a system. However, whereas the former must have bounded resource utilization, particularly in terms of its runtime where it is often on the fast-path, the performance of the latter code is instead measured in terms of the correctness and optimality of the resource allocation and configuration it produces, while its speed is less critical. As we have pointed out [26], these two areas of functionality are at present typically implemented in the same code base, inside the OS kernel, as low-level C code.

Our hypothesis is that the balance is tipping in favor of expressing configuration logic, and hardware configuration information, in a rich and high-level language. This enables complex resource allocation and configuration algorithms to be succinctly expressed, while being more amenable to adaptation due to changes in hardware technology, faulty hardware information ("quirks"), varying resource constraints and optimization goals, and device hotplug. Moreover, the same framework gives applications and user-level runtimes greater visibility into the available hardware resources and their current configuration.

We introduce three main contributions. First, in Section 2 we use PCI as an example to demonstrate the complexity of hardware configuration as an emerging issue in system software, and propose the use of declarative language techniques to mitigate its complexity as hardware becomes both more diverse and more complex.

Second, in Section 3 we describe in detail our initial approach to PCI bus configuration using the ECL^{i}PS^{e} constraint logic programming (CLP) system [2], a language with constraint-satisfaction extensions, and in Section 4 our solution to the related problem of interrupt allocation. We have implemented full PCI configuration and interrupt assignment for the Barrelfish [8] research operating, using the system knowledge base's (SKB) [26] CLP solver.

Finally, in Section 5 we present a combined evaluation of this work, focusing on its complexity, adaptability, and performance in comparison to the traditional approach, and in Section 6 discuss our experience with the new approach so far. The drawbacks include the need for a complex code base for the language runtime, and increased time to calculate configuration information. In exchange, the benefits include flexibility, efficiency of resulting configurations, conciseness of expression, and easy accommodation of special cases, and the ability to easily integrate extra information to guide resource allocation. We also discuss how trends in hardware and software are likely to affect this tradeoff.

2. Background: PCI allocation

Configuring the PCI bridges found in a typical modern computer is emblematic of a wide class of hardware-related systems software challenges: it involves resource discovery followed by allocation of identifiers and ranges from compact spaces of identifiers and addresses. More importantly, a range of hardware bugs and/or ad-hoc constraints on particular devices lead to a plethora of special cases which make it hard to express a correct algorithm in imperative terms. Worse, new hardware (whether system boards or devices) appears all the time, and system software must continue to work, or evolve to handle new cases with a minimum of disruptive engineering effort.

In this section, we describe the PCI programming challenge in detail. We start with the "idealized" problem, which appears relatively straightforward, and progressively introduce the complexities that, combined, are the reason that even modern operating systems only partially solve the full problem.

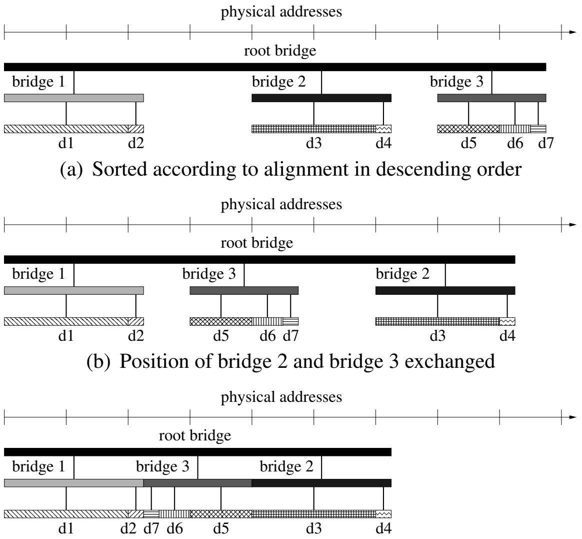
2.1 PCI background

A PCI (or PCI Express) interconnect is logically an n-ary tree whose internal nodes are *bridges* and whose leaves are *devices* [9, 23]. The root of the tree is known as the *root bridge* or *root complex*. Connections in the tree are known as *buses* (in legacy PCI they are electrically buses, whereas in PCI Express the bus is a logical abstraction over point-to-point messaging links). Non-root bridges are said to link *secondary buses* (links to child bridges and devices) to a *primary bus* (the link to the bridge's parent). High-end PCs often have two or four root complexes, and hence multiple PCI trees within a single system. Non-root devices can be attached to any bus in a PCI interconnect. Each device implements one or more distinct *functions*. A PCI function is in fact what we think of an independent "device" which has an own address represented by the bus number, the device number and the function number and which operates independently of other functions.

Driver software on host CPUs accesses PCI functions by issuing memory reads and writes or (in the case of the x86 architecture) I/O instructions. These requests are routed down the tree by the bridges, before being decoded by a single leaf device. Each function decodes a portion of the overall memory and I/O address spaces using a mapping that is configured by the host system through standard PCI-defined registers on each bridge and function.

Each "function" of a non-bridge device may decode up to 6 independent regions of either memory or I/O address space. These regions are defined and configured by *base address registers* (BARs) implemented by each function. The PCI driver queries each BAR to determine its required size, alignment, address space (memory or I/O), and, in the case of a memory-space BAR, whether the memory is *prefetchable* or *non-prefetchable*. Although it goes against strict PCI terminology, in the rest of this paper we will use "device" to denote a PCI function, i.e. a single logical device with up to 6 BARs.

Bridges also decode addresses to route requests between their parent and secondary buses. Unlike other devices, however, bridges use three pairs of base and limit registers instead of BARs, one each for prefetchable memory, non-prefetchable memory, and I/O space. Each bridge therefore decodes 3 independent, contiguous regions of IO or memory address space. The addresses used by every device below a bridge (including bridges on secondary buses) must lie within these three regions.



(a) Sorted according to alignment in descending order

(b) Position of bridge 2 and bridge 3 exchanged

(c) Devices under bridge 3 reordered

Figure 2. Alternative PCI configurations (only memory space resources are shown)

In summary, a host CPU accesses a PCI device by issuing a transaction on the system interconnect with a physical address that lies in a region decoded by the root bridge of the corresponding PCI tree. This is routed down the tree by bridges; at each level, each bridge on a bus compares the address issued by the CPU to the ranges defined by its base and limit registers. If it matches, the bridge forwards the request to its secondary bus. Each device on a bus compares the address to the regions defined by its BARs, and if the address matches, consumes it and generates a reply.

The PCI programming problem is to configure the base and limit registers of every bridge, and the BARs of every device function, to allow all the hardware registers for every device to be accessible from a CPU. As Figure 2 shows, this can be achieved in many different ways, leading to different usage of the available physical address space and different device locations in that space.

We can now specify the requirements for any PCI programming solution, starting with the basic properties of a solution in the "ideal" case, and progressively refining the list by adding real-world complications.

2.2 Basic PCI configuration requirements

Every bridge in a correctly-configured PCI tree decodes a subrange of the addresses visible on its parent bus. In order for all devices behind a bridge to be reachable, PCI requires that:

1. The *bridge window*, defined by its base and limit registers, must include all address regions decoded by all devices and bridges on the secondary bus.

In order that a request is forwarded by at most one bridge, sibling bridges sharing a bus must decode disjoint address ranges. Since a bus may contain both bridges and devices, all bridges and devices on a given bus must decode disjoint address ranges within the range of the parent bridge. This applies in all of the address spaces:

2. Bridges and devices at the same tree level (siblings) must not overlap in either memory or I/O address space.
3. The prefetchable and non-prefetchable memory regions decoded by a bridge or device must not overlap.

Regions of addresses in PCI must also be aligned. For a BAR, the base address must be "naturally" aligned at a multiple of the region's size. Similarly, a bridge's base and limit registers also have limited granularity, giving us the following alignment constraints:

4. BAR base addresses must be naturally aligned according to the BAR size.
5. Bridge base and limit register values for both memory regions must be aligned to 1MB boundaries.
6. Bridge base and limit register values for the I/O region must be aligned to 4kB boundaries.

These requirements constrain the possible locations of device BARs and child bridge base and limit registers within the region decoded by the parent bridge, potentially leading to gaps in address space for padding, as in Figures 2(a) and 2(b).

As described so far, configuring a PCI tree is a non-trivial problem, but can still be efficiently programmed by, for example, executing a post-order traversal of the PCI tree, sorting devices and bridges by descending alignment granularity, and allocating the lowest suitable address range in the appropriate address space at each step. Unfortunately, a number of real-world characteristics of modern computers, like for example the requirement to align addresses naturally, make this simple approach unworkable. Therefore, the simple post-order traversal results in a solution like that in Figure 2(a) where big padding holes need to be inserted between devices.

2.3 Non-PCI devices

The first problem is that certain non-PCI devices (e.g. IOAPICs and other platform devices) appear at fixed physical memory addresses, inside the region allocated to a PCI root complex. The locations of these devices may be discovered through platform-specific mechanisms such as ACPI [12], and no PCI device may decode such an address region.

7. Devices must not decode reserved regions of physical address space given by, for example, ACPI, or used by other known non-PCI devices such as IOAPICs.

2.4 Fixed-location PCI devices

Some PCI devices may be initialized and enabled by platform firmware at early boot time, for example USB controllers, network interfaces, or other boot devices. Naïvely reprogramming the BARs of such devices may lead to machine check exceptions or crashes since the device may be active, and performing DMA operations. Most operating systems avoid reprogramming the BARs of such devices, which means that their existing address assignment must be preserved. This further constrains the address ranges usable by parent bridges.

8. Certain PCI devices determined at boot cannot change location, and must retain addresses assigned to them by the BIOS.

2.5 Quirks

Hardware has bugs, and both devices and bridges can report incorrect information, fail to support valid resource assignments, or behave incorrectly when specific register values are programmed. These problems are known as PCI "quirks" and affect a wide range of shipping devices – the Linux 2.6.34 kernel lists 546 quirks – leading to a collection of workarounds in commodity operating systems. Common quirks include:

- devices that provide incorrect information about their identity as bridges or non-bridges;

- devices which decode more address range than advertised, or which decode address regions not assigned to them;
- standard devices which are hidden by platform firmware, but could otherwise be normally used;
- undefined device behavior (data loss on the bus, reduced bandwidth, system hangs, etc.) when particular values are written to configuration registers.

In the latter case, the PCI configuration process must ensure the problematic register values are never written, which imposes additional constraints on valid address assignments. Thus:

9. Configurations that would cause problematic values to be written to registers on specific devices must be avoided.
10. Incorrect information from PCI discovery must be corrected before calculating address assignment.

2.6 Device hotplug

Hotplugging, the addition or removal of PCI devices at runtime, raises another challenge. When a device is plugged in, the OS is notified by an interrupt from the root bridge, and must allocate resources to the BARs of the newly-installed device before it can be used. However, this may require reconfiguring and/or moving the address allocation of bridges and other devices in order to make enough address space available for the device, since it was not present at system boot. Changing the resource allocation of existing devices requires the driver to temporarily disable the device potentially save its current state first. After the new resources are programmed to the BARs, the driver needs to restart the device using the newly allocated resources. Depending on the device, it may need to bring the device to the saved state. This is a disruptive process and may not be supported by all devices, so the reallocation of resources which occurs on hotplug typically attempts to move the fewest possible existing devices and bridges.

11. Configuration should minimize the disruption caused by future hotplug events as much as possible.
12. Hotplug events should cause the minimal feasible reconfiguration of existing devices and bridges.
13. Hotplug-triggered reconfiguration may not move devices whose drivers do not support relocation of address ranges.

2.7 Discussion

It should by now be clear that PCI configuration is a somewhat messy problem characterized by a large (and growing) number of hardware-specific constraints which nonetheless have effects which propagate up and down the PCI tree. Consequently, most "clean" solutions written imperatively in a language like C sooner or later fall foul of an exception which can greatly complicate the code, compromise its correctness, reduce the efficiency with which it can manage physical address spaces, and in some cases prevent it from supporting the full PCI feature set.

Most current operating systems, including Linux [25, 29] and FreeBSD [5] on x86-based platforms, rely on platform firmware (BIOS or EFI) to allocate resources to most devices before the OS starts, and then run one or more post-allocation routines [4] to correct any problems in the allocation, allocate resources to devices left unconfigured by the firmware, and handle known quirks as devices are discovered and started.

This approach cannot guarantee success (though it often works): if a bridge is programmed with an address region that is too small to allocate all the devices behind it, there may be no way to grow the size of the bridge's address region without moving other bridges, and thus some devices behind the bridge will be rendered unusable despite sufficient address space being available overall. This problem is exacerbated by device hotplug, as it is impossible to predict at start-up the required size of all devices.

Even so, this simplistic allocation strategy leads to substantial code complexity: the complete PCI drivers of x86 Linux and FreeBSD account for approximately 10k and 6.5k lines of C code respectively, and device-specific quirks account for an additional 3k lines of code in Linux.

On other hardware platforms (such as Alpha/AXP), the firmware does not implement PCI configuration, and Linux instead performs a complete allocation using a greedy approach: devices are sorted by their requested size in ascending order, and resources allocated for each device in that order [25]. This can also lead to unusable devices behind a bridge, due to a suboptimal ordering of devices causing a shortage of address space. Note also that very little code is shared between this implementation and that for the PC platform: bug fixes or feature enhancements for one architecture may not be easily applied to another.

Until recently, Microsoft Windows used a similar strategy to x86 Linux and FreeBSD for PCI configuration, running a fix-up procedure to correct deficiencies in the firmware allocation. As with Linux and FreeBSD, this was unable to resize or change the address regions decoded by bridges, leading to potentially unusable devices [21]. Windows Vista and Server 2008 introduced a new re-balancing algorithm [20], allowing a bridge's resources to be modified according to the needs of its secondary bus, and increasing the likelihood that all PCI devices could be configured. However, this requires additional driver support for re-balancing, and the iterative approach can lead to highly complex multi-level re-balancing. Multi-level re-balancing might potentially need a long time, because increasing a bridge's window size may require to move the bridge to a new big enough free region. This may require more space from the parent bridge because of address alignment requirements, which again may be a problem. In the worst case, multi-level re-balancing may lead to a complete permutation of the PCI tree.

3. PCI resource allocation

The previous section detailed the PCI configuration problem and current approaches to solving it. In this section, we describe our implementation of PCI configuration in Barrelfish, and in the following Section 4, a solution to the closely-related problem of interrupt allocation, before evaluating both in Section 5.

Barrelfish [7,8] is a research operating system developed at ETH Zurich and Microsoft Research to address the related problems of scaling and system diversity in future heterogeneous multicore computers. As such, it provides a convenient testbed for our ideas.

PCI resource configuration can be viewed as a constraint satisfaction problem: for a given system, the variables are the base address allocated to each device BAR, and the base and limit of each bridge for each memory region it decodes, and a correct solution may be expressed as an assignment of integer values to these variables satisfying a series of constraints: alignment, sizes, and non-overlap of regions.

The difficulty in PCI resource allocation arises from satisfying these complex constraints. Such complexity is difficult to manage in a low-level systems language like C, but fortunately its runtime performance is not critical to the functioning of the system as a whole. This allows us the freedom to reformulate it in a declarative language, where the challenge becomes closer to defining *what* result we require, than *how* the result is to be produced.

We implemented the PCI resource configuration algorithm as a constraint logic program. This program operates on a high-level data structure representing the PCI tree, consisting of numeric variables and constraints between them that determine the possible solutions. Rather than worrying about how to allocate concrete addresses to bridges and devices, we instead concern ourselves with specifying the correct set of constraints to guide the CLP solver. We begin by describing the separation between C and CLP code, before explaining the constraint logic in detail.

3.1 Approach

We explicitly separate the PCI configuration algorithm, expressed in CLP and running in a user-space service, from the register access and device programming mechanisms, implemented in the usual C code as part of the PCI subsystem of the OS. This has several advantages. First, it decouples the details of the configuration algorithm from the device access code, allowing us to exchange and evolve the algorithm independently of the device access mechanisms. Second, the algorithm is expressed only in terms of the generic PCI bus – all architecture-specifics are confined to the device access code.

The CLP solver we use is the Barrelfish system knowledge base [26], a port of the open-source ECL^{i}PS^e CLP system. The SKB is a service which is started early in the Barrelfish boot sequence and runs initially from a RAM disk image, enabling it to function without any device support. It is passive and event-driven, responding to requests from the PCI driver.

At start-up, the PCI driver performs device discovery. The location of root bridges is determined by platform-specific mechanisms such as ACPI [12]. The driver then walks the entire bus hierarchy, determining the complete set of bridges, devices and BARs that are present, and assigning bus numbers to un-numbered bridges. As part of this pass, the PCI driver inserts Prolog *facts* in the SKB. Those facts describe the set of present bridges, devices and BARs, according to the following schema:

```
rootbridge(addr(Bus, Dev, Fun), childbus(MinBus, MaxBus),
    mem(Base, Limit)).
bridge(pcie | pci, addr(Bus, Dev, Fun), VendorID, DevID,
    Class, SubClass, ProgIf, secondary(BusNr)).
device(pcie | pci, addr(Bus, Dev, Fun), VendorID, DevID,
    Class, SubClass, ProgIf, IntPin).
bar(addr(Bus, Dev, Fun), BARNr, Base, Size, mem | io,
    prefetchable | non-prefetchable, 64 | 32).
```

These facts encode all information needed to run the PCI configuration algorithm. A root bridge is identified by its PCI configuration address (bus, device and function number), the range (minimum and maximum) of bus numbers of its children, and its assigned physical memory region. Bridges and devices are identified by their address, and carry standard identifiers for their vendor, device ID, device class and subclass, and programming interface. A bridge also includes the bus number of its secondary bus, and a device the interrupt pin which it will raise (which is used by the interrupt allocation routines described in Section 4). Finally, for a BAR we store its base address (which may have been previously assigned by firmware), required size, region type, and whether it is a 64-bit or 32-bit BAR.

After creating the facts, the PCI driver causes the SKB to run the configuration algorithm to compute a valid allocation. The initialization algorithm we use is described in the following section. Its output is a list of addresses for every device BAR and every bridge, which can be directly programmed into the corresponding registers by the driver. For example:

```
buselement(device, addr(6,0,0), 0, C0000000, D0000000,
        10000000, mem, prefetchable, pcie, 64),
buselement(bridge, addr(0,15,0), secondary(6), B0100000,
        D0000000, 1FF00000, mem, prefetchable, pcie, 0)
```

In this example, the 64-bit PCIe device at bus 6, device 0, function 0 requests a physical address range of 256MB in prefetchable memory space for BAR 0. The base allocated to the device is 0xC0000000 and the limit will thus be 0xD0000000. The bridge at which the device is attached has a base of 0xB0100000 and a limit of 0xD0000000 in the prefetchable memory space, clearly including this device (along with others, not shown here).

The PCI driver takes the addresses and BAR numbers as well as bridge base and limit values from the output, and programs the specified registers. While reprogramming devices and bridges, they are disabled to prevent transient address conflicts. Once reprogramming is complete, the bus is completely configured and device drivers can be started.

3.2 Formulation in CLP

We now turn to the configuration algorithm in constraint logic. Its first step is to convert the facts generated by the PCI driver to a suitable data structure, and declare the necessary constraint variables. The data structure used is a tree mirroring the hardware topology, whose inner nodes correspond to bridges, and leaf nodes to device BARs or other unpopulated bridges. The constraints are then naturally expressible through recursive tree traversal. The variables of the CLP program are the base address, limit and size of every bridge and device BAR, and the relationship between them may be expressed by the constraint `Limit $= Base + Size`, which we later apply.

At a high-level, our algorithm performs the following steps for each PCI root bridge:

1. Convert `bridge` and `device` facts for the given root bridge to a list of `buselement` terms, while declaring constraint variables for the base address, limit and size of each element.
2. Construct a tree of `buselement` terms, mirroring the PCI tree.
3. Recursively walk the tree, constraining the base, limit and size variables according to the PCI configuration rules and quirks.
4. Convert the tree back to a list of elements.
5. Invoke the ECL^{i}PS^e constraint solver to compute a solution for all base, limit and size variables satisfying the constraints.

The core logic of the algorithm resides in step 3 above, and we implement this by a direct translation of the rules described in Section 2.2 to constraint logic, as described in the following sections.

Bridge windows

Rule 1 states that all bridge windows must include all address regions decoded by devices and bridges attached to the secondary bus. This means that the bridge's memory and IO base addresses must be smaller or equal to the smallest base of any bridge or device on the secondary bus, and the corresponding limits must be greater than or equal to the highest address used by any device or bridge on the secondary bus.

Although we do not yet have concrete values for the relevant base and limit variables, CLP allows us to constrain them using a recursive walk of the tree, implemented as shown below.
Note that a tree is expressed as `t(Root,Children)`, where `Root` is the root node, and `Children` is a (possibly empty) list of child trees – ECL^{i}PS^e uses conventional Prolog syntax whereby identifiers starting with an uppercase character (e.g. `Node`) denote free variables, and all others denote constants. Also note the ECL^{i}PS^e operations `ic_global:sumlist`, `ic:minlist` and `ic:maxlist` which operate on lists of constraint variables that may not have a concrete value assigned, allowing complex constraints to be introduced between them.

```
setrange(Tree,SubTreeSize,SubTreeMin,SubTreeMax) :-
  % match Tree into current node and list of children
  t(Node,Children) = Tree,
  % match node to get its base, limit and size variables
  buselement(_,_,_,Base,Limit,Size,_,_,_,_) = Node,

  % recursively collect lists of sizes, minimum and
  % maximum addresses for children of this node
  ( foreach(El,Children),
    foreach(Sz,SizeList),
    foreach(Mi,MinList),
    foreach(Ma,MaxList)
    do
      setrange(El,Sz,Mi,Ma)
  ),

  % compute sum of children's sizes as SizeSum
  ic_global:sumlist(SizeList,SizeSum),
  % constrain the size of this node >= SizeSum
  Size $>= SizeSum,

  % if there are any children...
  ( not Children=[] ->
      % determine min base and max limit of children
      ic:minlist(MinList,Min),
      ic:maxlist(MaxList,Max),
      % constrain this node's base and limit accordingly
      Min $>= Base,
      Max $=< Limit
    ; true
  ),

  % constrain this node's limit
  Limit $= Base + Size,

  % output values
  SubTreeSize $= Size,
  SubTreeMin $= Base,
  SubTreeMax $= Limit.

setrange([],0,_,_). % base case of recursion
```

Non-overlap of bridges and devices

Rule 2 states that siblings must not overlap at any level of the tree. In other words, all regions allocated to bridges and devices at the same level must be disjunctive. The following goal ensures this, by making use of the disjunctive constraint, which ensures that regions specified as lists of base addresses and sizes do not overlap:

```
% convenience functions / accessors
root(t(R,_),R).
base(buselement(_,_,_,Base,_,_,_,_,_,_),Base).
size(buselement(_,_,_,_,_,Size,_,_,_,_),Size).

nonoverlap(Tree) :-
  % collect direct children of this node in ChildList
  t(_ ,Children) = Tree,
  maplist(root,Children,ChildList),

  % if there are children...
  ( not ChildList=[] ->
      % determine base and size of each child
      maplist(base,ChildList,Bases),
      maplist(size,ChildList,Sizes),

      % constrain the regions they define not to overlap
      disjunctive(Bases,Sizes)
    ; true
  ),

  % recurse on all children
  ( foreach(El, Children) do nonoverlap(El) ).
```

Non-overlap of prefetchable/non-prefetchable memory

Rule 3 requires that prefetchable and non-prefetchable regions do not overlap. The two regions do not need to be contiguous. Therefore we implemented this by inserting an artificial level in the top of the tree containing two separate bridges, one with all prefetchable memory ranges and another with all non-prefetchable memory ranges of the tree. This gives some freedom to the solver, because the order of the two regions is not explicitly specified by our allocation code, and allows the previously-described logic to operate independently of memory prefetchability. Treating the two regions as completely separate trees causes the prefetchable and non-prefetchable window of every bridge to be at completely different locations, which is fine.

Alignment constraints

Rules 4, 5 and **6** require a specific alignment for devices and bridges. In the following, we constrain the alignment of each element, using natural alignment for device BARs, and a fixed alignment for bridge windows (e.g. 1MB in the case of memory regions).

```
naturally_aligned(Tree, BridgeAlignment, LMem, HMem) :-
  t(Node,Children) = Tree,

  % determine required alignment for bridge or device BAR
  ( buselement(device,_,_,Base,_,Size,_,_,_,_) = Node ->
      Alignment is Size; % natural alignment
    buselement(bridge,_,_,Base,_,_,_,_,_,_) = Node ->
      Alignment is BridgeAlignment % from argument
  ),

  % constrain Base mod Alignment = 0
  suspend(mod(Base, Alignment, 0), 0, Base->inst),

  % recurse on children
  ( foreach(El, Children),
      param(BridgeAlignment), param(LMem), param(HMem)
    do naturally_aligned(El, BridgeAlignment, LMem, HMem)
  ).
```

Reserved regions

Rule 7 requires that reserved memory regions are not allocated to PCI devices. In other words, memory regions allocated to PCI devices should always be disjunctive with any reserved region. The following goal ensures this requirement, by recursively processing a list of bus elements against a list of reserved memory ranges, specified as range(Base,Size) terms:

```
% recursive stopping case
not_overlap_mem_ranges([], _).

% bridges may overlap: no special treatment
not_overlap_mem_ranges(
  [buselement(bridge,_,_,_,_,_,_,_,_,_)|T], MemRanges) :-
    not_overlap_mem_ranges(T, MemRanges).

% device BARs match this pattern
not_overlap_mem_ranges([H|T], MemRanges) :-
  % for each reserved memory range...
  ( foreach(range(RBase,RSize),MemRanges), param(H)
    do
      % match base and size variable from bus element
      buselement(device,_,_,Base,_,Size,_,_,_,_) = H,
      % constrain this BAR not to overlap with it
      disjunctive([Base,RBase], [Size,RSize])
  ),
  % recurse on list tail
  not_overlap_mem_ranges(T, MemRanges).
```

Fixed-location devices

We must also avoid moving various initialized boot devices, as in **rule 8**. The following goal shows one such example: given a device class (specified by its class, subclass and programming interface identifiers) that should not be moved, it constrains the possible choice of the base address to the one value which is its initial allocation.

```
keep_orig_addr([], _, _, _).
keep_orig_addr([H|T], Class, SubClass, ProgIf) :-
  ( % if this is a device BAR...
    buselement(device,Addr,BAR,Base,_,_,_,_,_,_) = H,
    % and its device is in the required class...
    device(_,Addr,_,_,Class, SubClass, ProgIf,_),
    % lookup the original base address of the BAR
    bar(Addr,BAR,OrigBase,_,_,_,_) ->
      % constrain the Base to equal its original value
      Base $= OrigBase
    ; true
  ),
  % recurse on remaining devices
  keep_orig_addr(T, Class, SubClass, ProgIf).
```

3.3 Quirks

Declarative logic programming provides an elegant solution to the problem of quirks. Quirks require us to correct wrong information as well as apply possible extra constraints to workaround misbehaving devices. In CLP we can easily define a database of facts for devices needing special treatment. Those facts are implicitly matched against the data structure before the configuration algorithm runs, causing incorrect information to be corrected, and additional constraints on the allocation to be defined, without changing any of the core logic of the algorithm.

3.4 Device hotplug

In principle, the allocation of resources for hotplugged devices can be handled simply by adding facts for the new device and its BARs, and then re-running the allocation algorithm. However, this may cause all existing address assignments to change (excluding those whose location is fixed, as in Section 3.2), and is thus undesirable due to the performance impact of interrupting running device drivers.

Adding artificial devices to the PCI tree before computing the first allocation helps to produce gaps which can later be used for hotplugged devices. Figure 3 shows that the CLP solution can deal with an almost completely filled region. This means, that the available space can almost be filled completely with artificial devices to provide space for later hotplugs. With CLP this is particularly easy, because the artificial devices get placed around the real ones. Later, when a device gets hotplugged, the algorithm should try to move the least possible number of BARs of other devices. CLP allows to define an objective function for the constraint solver, minimizing the number of BARs which have to be reallocated. Moreover, the CLP solution is better placed to handle complex reconfiguration that may be required by device hotplug, as it specifies the complete set of feasible configurations which will be explored by the solver. Section 5.4 presents the results of a benchmark showing the theoretical limits of the CLP approach in handling device hotplug, in comparison to a traditional postorder traversal.

4. Interrupt allocation

We now move from PCI bus configuration to the closely-related problem of interrupt allocation, which we have also implemented in CLP, and which is also evaluated in Section 5.

4.1 Problem overview

Interrupts are another important resource that must to be allocated to devices by the OS. Most PCI devices can raise one or more interrupts. To avoid shared interrupt handlers, the OS should try to allocate unique interrupt vectors to every device. Modern systems, and some modern devices, support message signaled interrupts (MSIs). These map interrupts into the physical address space, and therefore the only requirement is choosing a different interrupt address for every device. However many systems and many PCI devices do not yet support MSIs, and thus correctly and efficiently configuring PCI interrupt allocation remains a critical OS task.

Each PCI device signals interrupts by asserting one of up to four available interrupt lines (INTA, INTB, INTC and INTD, represented in our solution as the integers 0–3). On PC-based platforms, these signals are routed via PCI bridges and configurable *link devices* to global system interrupt numbers (GSIs). This routing is encoded in and configured via platform firmware, using a set of ACPI tables [12].

Starting from a given device and interrupt pin, the mapping is determined as follows:

1. Consult the ACPI interrupt routing tables for the current bus, device and pin number. If there is a mapping for the given pin:
   (a) If the entry names a GSI, the interrupt line is fixed.
   (b) Otherwise, the entry names a link device, and the interrupt is selectable from set of GSIs.
2. Otherwise, compute the new interrupt pin on the parent bus, using the formula (*device number* + *pin*) mod 4, and repeat.

The goal of the interrupt allocation code is to assign unique interrupt vectors to every device. Interrupt sharing is to be avoided wherever possible [14]. It can severely impact performance, since the drivers for devices sharing an interrupt must essentially poll their devices to determine if the interrupt is for them. Furthermore, many device drivers do not handle shared interrupts correctly at all. As well as avoiding sharing among PCI devices, specific GSIs are also assigned to legacy (non-PCI) devices and other system devices, which should also be avoided by the allocation code.

We can summarize the requirements for interrupt configuration as follows:

1. assign and configure a GSI (possible translated by bridges and link devices) for every enabled PCI device,
2. ensure that all allocated GSIs are unique.
3. avoid reassigning legacy pre-allocated GSIs.

This problem is not as hard as the PCI address allocation problem and could be implemented in C. However there are still some benefits from using CLP. First, storing and querying information about possible GSIs and prototyping the algorithm in CLP is convenient. Second, ensuring that allocated GSIs are globally unique can easily be done using the built-in goal `alldifferent` (see 4.2). We therefore implemented interrupt allocation in the SKB.

4.2 Solution in CLP

At start-up, the PCI/ACPI driver code populates the system knowledge base with a fact for every PCI interrupt routing table entry, mapping a device address and interrupt pin to a source, using the schema:

```
prt(addr(Bus, Dev, _), Pin, pir(Pir) | gsi(Gsi)).
```

These facts include addresses of PCI devices without function number, because the same mapping applies for all functions on a multi-function device. The interrupt source is either a name (ACPI object path) identifying the interrupt link device or a direct GSI number, indicating that this interrupt's allocation is fixed.

For each link device, pir facts are added describing the possible GSIs that may be selected for a given device:

```
pir(Pir, GSI).
```

In this relation, Pir defines the link device name, and GSI one of the selectable GSIs for this device (so each link device has multiple facts, one for each configuration).

The CLP code operates on these facts, and the PCI device facts described in the previous section. At the top-level, it determines the interrupt pin used by a specific device, and passes it to assignirq to allocate a unique GSI:

```
assigndeviceirq(Addr) :-
  device(_, Addr, _, _, _, _, _, Pin),
  % require a valid Pin
  Pin >= 0 and Pin < 4,
  ( % check for an exising allocation
    assignedGsi(Addr, Pin, Gsi),
    usedGsi(Gsi, Pir)
  ; % otherwise assign a new GSI
    assignirq(Pin, Addr, Pir, Gsi),
    assert(assignedGsi(Addr, Pin, Gsi))
  ),
  printf("%s %d\n", [Pir, Gsi]).
```

assignirq takes the PCI address and interrupt pin for the device as inputs, and chooses a possible GSI for the device. It uses findgsi (described below) to determine the available GSIs for the device, and the alldifferent goal to avoid overlaps:

```
assignirq(Pin, Addr, Pir, Gsi) :-
  % determine usable GSIs for this device
  findgsi(Pin, Addr, Gsi, Pir),
  ( % flag value for a fixed GSI (i.e. meaningless Pir)
    Pir = fixedGsi
  ;
    % don't change a previously-configured link device
    setPir(Pir, _) -> setPir(Pir, Gsi)
  ;
    true
  ),
  % find all GSIs currently in use
  findall(X, usedGsi(X,_), AllGsis),
  % constrain GSIs not to overlap
  ic:alldifferent([Gsi|AllGsis]),
  % allocate one of the possible GSIs
  indomain(Gsi),
  % store settings for future reference
  ( Pir = fixedGsi ; assert(setPir(Pir,Gsi)) ),
  assert(usedGsi(Gsi,Pir)).
```

Finally, the following CLP function matches the device's address and interrupt pin with the prt and pir facts to find the possible GSIs (multiple solutions may be found). If no match is found, it recursively performs bridge swizzling until a routing table entry matches (which is always true at the root bridge).

	Devices	BARs	Bridges	Runtime (ms)
sys1	7	11	12	2.0
sys2	13	20	6	14.7
sys3	13	20	6	14.4
sys4	14	22	6	36.4
sys5	12	18	5	10.0
sys6	7	9	6	19.0
sys7	9	14	6	22.2
sys8	15	25	4	6.7
sys9	15	25	4	31.2

Table 1. System complexity and execution times for the PCI configuration algorithm

```
findgsi(Pin, Addr, Gsi, Pir) :-
  ( % lookup routing table to see if we have an entry
    prt(Addr, Pin, PrtEntry)
  ;
    % if not, compute standard swizzle through bridge
    Addr = addr(Bus, Device, _),
    NewPin is (Device + Pin) mod 4,

    % recurse, looking up mapping for the bridge itself
    bridge(_, BridgeAddr, _, _, _, _, _, secondary(Bus)),
    findgsi(NewPin, BridgeAddr, Gsi, Pir)
  ),
  ( % is this a fixed GSI, or a link device?
    PrtEntry = gsi(Gsi),
    Pir = fixedGsi
  ;
    PrtEntry = pir(Pir),
    pir(Pir, Gsi)
  ).
```

5. Evaluation

Picking suitable metrics to evaluate a PCI programming solution is something of challenge. We focus here on code complexity, execution time, and efficiency of resultant solutions, but some of the evaluation necessarily remains subjective in its comparison with current approaches.

5.1 Test platforms

We evaluated the PCI configuration and interrupt allocation algorithms on nine different x86 PC and server systems, with a mixture of built-in and expansion devices including network, storage and graphics cards installed. We refer to these as sys1 through sys9, and show the number of PCI elements they include in Table 1. All systems have two PCI root bridges with the exception of sys1, which has one. Here we show the totals for the whole system, as our algorithm allocates resources to all PCI trees in a single invocation.

All of these systems support USB keyboards in the BIOS, and thus the system initializes the USB controller in firmware at boot time. Consequently, our solutions implement this fixed device requirement using the keep_orig_addr constraint from Section 3.2 to prevent the USB controllers from being reprogrammed, and also avoid any memory regions marked as reserved by ACPI or in use by IOAPIC devices. The computation does not include handling other quirks, since our hardware does not exhibit them and consequently does not exercise that part of our CLP code. Our implementation is successful in configuring all PCI buses and devices on all the test systems.

	C LOC	CLP LOC
Register access	235	
Data structure	817	31
Algorithm		224
ACPI	360	
Interrupts	660	28
Miscellaneous	109	
Total	2181	283

Table 2. Lines of code in PCI configuration and interrupt allocation

	C LOC
Register access	897
Data structure	1686
Resource management	706
ACPI	121
Interrupts	521
Miscellaneous	45
Total	3976

Table 3. Lines of code for equivalent functionality in Linux

5.2 Performance

We measured the time for PCI configuration on our test systems, and show the results in Table 1. This time is for the CLP algorithm and does not include the initial bus walk, nor programming of device registers. As discussed in Section 3, these remain in C as part of the PCI driver, and the CLP time dominates the overall runtime.

Compared to the performance of a hard-coded allocation in C, which in existing OSes typically requires less than a millisecond, our solution is substantially slower, but the additional overhead of 10–30ms is only incurred at boot time or after a hotplug event, and so is arguably insignificant to the end user. This computation can be run in parallel with other tasks, and since the PCI configuration changes rarely, the computed configuration can be cached and reapplied during the next boot process. In those cases, no additional overhead is added to the boot time.

5.3 Code size

In this section we compare the complexity, measured in lines of code (LOC), of our CLP-based approach to the comparable portions of the Linux x86 PCI driver. Such a comparison can never be precise, and must be preceded by several qualifications. First, in both cases we consider the code related to PCI resource configuration, interrupt allocation, PCI device discovery, maintenance of the data structures representing the PCI bus hierarchy, and the corresponding hardware access mechanisms. Second, we exclude some PCI-related mechanisms (such as the legacy PCI BIOS interface) that are currently unsupported by our solution. Third, because we do not handle quirks yet, we exclude the hardware quirk-handling code, but retain handling of other special cases. Finally, the Linux code implements the solution that attempts to fix up the initial BIOS configuration, whereas our code does a full allocation of addresses. Note that our goal is to reduce the complexity of the source code and therefore the number of source lines of code, rather than the number of generated machine statements.

We summarize the results for our solution in Table 2 and for Linux in Table 3. The relevant Linux code is located in the kernel in drivers/pci. Overall, our approach uses 2464 lines of code, compared to 3976 for the pure C-based Linux version.

Breaking this down, we use much less code for register access, as our access to hardware is highly regular and independent of allocation. Building and manipulating data structures is also simpler for us: representing lists and trees is highly concise in Prolog, and allows us to build much simpler structures in the C domain, resulting in about half the code size. We use more code for ACPI, since we explicitly handle ACPI reserved regions, whereas Linux relies on the BIOS initialization for this. Code for interrupt assignment is about the same size. Finally, the "core" of the configuration code (in as much as it can be isolated in the Linux case) is 224 lines of Prolog versus 706 lines of C.

The largest class of code in both implementations is used for maintaining data structures. This is because PCI data must be queried from either ACPI or directly from the hardware, transformed to a meaningful internal representation, and added to a structure. Finally, configuration proceeds by traversing this structure, accessing and mutating it. The corresponding data structure in our implementation consists mostly of Prolog facts which are generated by C but traversed/accessed entirely in CLP, and thus require fewer lines of code than Linux. Despite being large in size in both systems, such code is not the most complex in its logic.

The more complex logic in the PCI configuration algorithm uses 224 lines of CLP code in our implementation. This not only produces a correct and complete allocation, it also handles special constraints such as avoiding reserved regions as well as not moving certain devices. In comparison, the Linux C implementation uses more lines of code for less functionality (it does not perform full bus configuration).

Besides the usual benefits arising from a smaller, simpler, codebase in terms of source lines of code, the separation of concerns between low-level hardware-specific device access code and a high-level declarative resource configuration algorithm enhances the system's maintainability and adaptability to changing hardware requirements. Complex device- and system-specific constraints, such as quirks, can be incorporated without changing the device access code or core algorithm, and it can easily be ported to other PCI-based platforms. We return to this discussion in Section 6.

5.4 Postorder traversal comparison

To evaluate the quality of the solutions found, we investigated how they compare to the style of simple postorder traversal used in current operating systems. When allocating resources to a device tree where the size of each device is known in advance, one might expect this approach to be sufficient. We first describe why that is not the case, and then show experimentally the advantage of a declarative CLP solution against such a traversal.

Starting with the base address given by the root bridge, such an algorithm traverses down the left-most branch of the tree first, assigning the current base address to each bridge and finally the left-most leaf device, while satisfying alignment constraints. For each device allocation, the device size is added to the base value, plus any padding required for alignment. The algorithm next traverses all child devices of the bridge, before moving up the tree to the next-upper parent bridge, and updating the bridge's limit register in the process, before continuing with the remaining devices and bridges.

Such an algorithm can be simply described and implemented, and ensures that all bridges are allocated a window including their children and that alignment constraints are satisfied. However, the algorithm is insufficient for PCI configuration for two reasons:

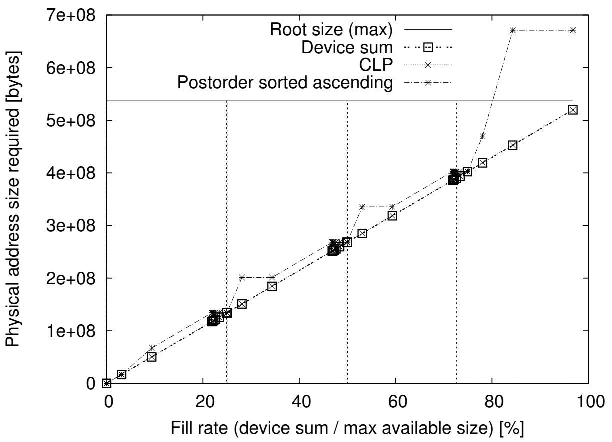


Figure 3. Address space utilization of CLP algorithm vs. simple postorder traversal as devices and bridges are added to a simulated system. The CLP algorithm reorders devices as needed, exactly following the *DeviceSum* line, which shows the lower bound. The postorder traversal, which sorts the devices according to size, cannot fit the PCI tree into the given root bridge window. Vertical lines indicate when a new bridge is added; the horizontal line indicates the maximum size of the root bridge window.

1. It fails to include constraints that require keeping devices at a fixed address. This requires all parent bridges to decode the fixed device window. Because all parent bridges have to decode a fixed address, all children of every bridge decoding a fixed address have to be placed close to a predetermined address region. This cannot be easily expressed in a postorder traversal of the device tree.
2. Satisfying alignment constraints leads to potentially large amounts of address space wasted in padding, preventing successful configuration when not all devices fit into the root bridge's address range.

To learn how or CLP-based algorithm behaves when resources are consumed more and more by additional devices, we stressed the configuration algorithm in an offline experiment by adding progressively more devices and bridges to a simulated PCI system. Starting with zero devices and bridges, we added either a device or a bridge on every round and measured the consumed resources by the configuration derived by the algorithm. This scenario is not purely artificial, because it simulates what can happen when devices are hotplugged. We compared our CLP-based algorithm with an improved postorder traversal algorithm, which sorts devices according to their requested size in ascending order. The results are shown in Figure 3.

The horizontal line *Root size (max)* indicates the given root bridge window size, which must not be exceeded for a successful configuration. The vertical lines in the figure indicate where a bridge has been added to the PCI tree. The *DeviceSum* line indicates the sum of the requested size of all installed devices without padding or alignment constraints; this is the absolute lower bound of address space utilization. The data points indicate the address space consumption after having added the next device.

The figure shows that our CLP-based allocation algorithm exactly follows the device sum. Its constraints give it the freedom to reorder bridges and devices, so that no address space is wasted for alignment constraints and a solution can always be found. The best postorder traversal algorithm, which does not respect fixed device requirements, cannot fit the devices into the given root bridge window beyond 80% utilization, indicating that such a simple approach cannot work in general.

6. Discussion

We set out to evaluate declarative languages as a way of expressing hardware configuration algorithms, as part of a wider project to build a new operating system for heterogeneous multicore systems. Our hypothesis was that such an approach would reduce the complexity of the code we would have to write, and in the long term would provide a good foundation for reasoning about the complexity and heterogeneity of modern and future hardware.

Our experience so far has been mostly positive, but not without challenges. In this section we describe both the advantages and disadvantages of the approach that we have encountered.

6.1 Advantages

Clear policy/mechanism separation: Maintaining a sharp distinction between the algorithm used to find a suitable hardware configurations and the mechanism to configure the hardware by writing values to registers has a number of strong benefits.

First, the algorithm can be clearly understood in isolation from hardware access, making it easier to both debug and maintain. Indeed, much of the debugging, testing, and evaluation of our PCI programming code was carried out "offline" in a vanilla ECL^{i}PS^{e} running on Linux using PCI configurations obtained from a variety of machines around our lab, before being put into service at boot time in Barrelfish. It is also useful to be able to test this code by writing correctness conditions in Prolog which are then validated automatically.

Second, the hardware access code is simplified, since it is no longer threaded through the configuration algorithm. Verifying (by inspection) that the C code correctly accesses PCI devices and bridges becomes a simpler task, and the chances of breaking this code when changing the configuration algorithm itself are greatly reduced.

Separation of special cases: PCI quirks, fixed PCI devices, reserved non-PCI address ranges, and the like can be handled entirely in the declarative domain through Prolog statements, and do not pollute the C register access code.

Furthermore, adding new quirks or special cases can be done simply by adding such cases as assertions to the declarative specification of the algorithm, *without* modifying the mainline algorithm code in any way. For the most part, additional constraints are one-line references to existing functions, and hence easy to add to the system. It is often sufficient merely to add a device's ID to a list, which is passed to a function applying a specific constraint to the elements. All of this results in a clear separation within the declarative code between special cases and the solution description.

Flexibility of data structures: Device information in traditional operating systems is typically represented by a set of simple, ad-hoc data structures (tables, trees, hash tables) whose design is determined largely (and rightly so) by performance concerns in the kernel. In our approach we retain such structures where needed on the fast path, but represent most of the hardware information as facts in the logic language.

This greatly facilitates reasoning about the information in ways not foreseen at design time. For example, information from ACPI about non-PCI device locations can be transformed easily into regions of memory reserved from the normal PCI allocation process. The logical unification mechanism provided in languages like Prolog makes this expressible in a single rule. Furthermore, this representation can be changed over time without concern for disturbing critical kernel code.

Late-binding of algorithm: ECL^{i}PS^e allows for adding new functionality as well as replacing functionality at runtime. This feature provides considerable flexibility. In the concrete case of PCI programming, we can run the normal allocation algorithm for a complete allocation and later at runtime load an allocation algorithm which is more suited to hotplug scenarios. Whether the algorithm is replaced at runtime or boot time, the mechanism code to access the hardware need not change.

Platform independence: As we have mentioned, PCI code in Linux varies almost completely between, for example, the x86 and Alpha/AXP platforms. In contrast, with our approach the configuration logic is identical across all architectures using PCI. What changes is the register access code in C – for example, most non-x86 architectures replace I/O instructions with memory-mapped I/O. This makes our code highly portable. Furthermore, only short mechanism code has to be ported, reducing the chance of introducing bugs when porting.

Reuse of functionality: While CLP may be regarded as a somewhat heavyweight approach (see below), the functionality provided is close to that required by many other parts of a functional OS – in some ways, the system knowledge base might be regarded as analogous to parts of the Windows Registry, albeit with a much more powerful type system, data model, and query language. Barrelfish uses this functionality for, among other things, representing the memory hierarchy to performance-conscious parallel applications, and as a name server for other system services. Along with the authors of Infokernel [3], we argue for making a rich representation of system information available for online reasoning, and CLP provides a powerful tool for achieving this.

6.2 Disadvantages

Unsurprisingly, the approach also has some significant drawbacks.

Constraint satisfaction is no silver bullet: A hardware configuration problem like PCI, with all its special cases, is very natural to express as a constraint satisfaction problem. However, this does not automatically lead to a solution in a reasonable time. Constraint solvers have a well-known tendency to explode in complexity (and, consequently, time of execution) without careful specification of the problem, and our use of CLP is no exception in this regard.

Part of this is due to ECL^{i}PS^e being a relatively simple solver by modern standards, but much of the complexity is inherent. In practice, the onus is on the programmer to guide the solver by careful annotation of the problem. This makes the source code more complex than a simple specification of the constraints - our Prolog code is carefully written to avoid an explosion in complexity and/or runtime.

For example, in our PCI case we sort the variables to be instantiated according to the requesting size of the device in an ascending order. The solver starts probing the last element of the list of variables. This causes it to try to place the device with the biggest size requirement first, which is generally more difficult. If small devices would be placed first, the solver would most likely later need to reallocate them, to free up a large continuous address range in order to place a bigger device. This would potentially lead to a whole permutation of the tree.

To take another example, the natural-alignment property is best expressed by a modulo division of the base address by the size, as shown in Section 3.2. If the remainder is zero, the address is aligned according to the size. However using this implementation, when the solver tries to instantiate the base address variable, it searches all integers one by one until it finds a value with a zero remainder. In case some other constraints cannot be met, the solver must try another solution and repeat searching for values with a remainder of zero, leading to a huge execution time. We therefore modified the associated goal slightly, by letting the solver choose an integer from a (typically small) precomputed range which is multiplied by the device's size to determine the base address. The upper bound of the range is chosen so that the maximum base lies just beyond the fixed window of the root bridge, therefore including all possible naturally-aligned base addresses for the device, while substantially reducing the search space.

Increased resource usage: Even with the heuristics described above, ECL^{i}PS^e CLP is an interpreted, high-level language with high execution time overhead compared to C. Additionally, a CLP algorithm works by propagating constraints and then probing values rather than assigning values in a straight-forward iterative way. Clearly this leads to longer execution times.

For some classes of problem, such as the PCI programming case we discuss in this paper, the execution time overhead is not critical as long as it remains under a second or so. Additionally, the PCI configuration changes rarely and the previous solution can be cached and reapplied without the need of starting the CLP system. In general, boot time is only increased when the PCI configuration changed since the algorithm ran last time. However, the performance penalty clearly rules out a class of other, time-critical hardware-related tasks.

For this work we used a relatively slow language runtime. Using a faster language might significantly improve execution time.

For resource constrained devices such as small battery powered sensor nodes or embedded systems this solution might be inappropriate. However those systems usually have a simple and fixed PCI configuration without hotplug support. They can either be programmed in C or by running the solver once, offline, on a standard PC with PCI data from the embedded system. The solution found can then be added to the device's boot image and applied at every boot-up of the system.

Large code base: While we use considerably less PCI-specific code (C and Prolog) to implement our solution, we do employ a large body of code in the form of the CLP solver. The port of ECL^{i}PS^e we use in Barrelfish consists of 16242 lines of C[1], plus a handful of assembly-language lines. In addition, the core CLP libraries add 1042 lines of Prolog, many of them quite long. The complete solver executable (statically linked) consists of 1.5MB for a 64-bit x86 OS. Additionally, a compressed RAM disk of 600kB provides the necessary Prolog files. This is clearly significant, and adding this amount of code to the boot image of an OS raises at least two concerns.

First, there is the issue of code bloat. On modern hardware, the boot process is unduly impacted by the overhead, but the difference in start-up performance is noticeable compared with the (considerably less functional) hard-coded PCI solution we used in the early stages of OS development. On the other hand, as mentioned above, the CLP solver does provide a valuable data management service to other parts of the OS as a general name server and policy engine, and so the cost in code size should be amortized over the whole set of client subsystems which use it.

Second, there is the extent to which we can trust the CLP solver itself. We rely on ECL^{i}PS^e behaving correctly. Since it is a mature, general-purpose system, we might expect it to be reliable and relatively bug-free. However, it is unlikely that a complex piece of code like ECL^{i}PS^e will be formally verified, which makes our approach less attractive for high-assurance operating systems. However, such systems typically are written to specific hardware platforms, obviating the need for complex configuration logic.

For high-assurance, formally verified systems, a better application of this approach would be to apply the ideas at compile time,

[1] LOC counts were generated using "SLOCCount" by David A. Wheeler.

which would integrate with the seL4 approach [17] of modeling the entire OS in a high-level language, which is then translated (in a way that preserves the verified properties) to C.

Boot sequence: Configuring hardware at OS boot time in a high-level language like CLP means that the language runtime has to be started early in the boot process. Barrelfish may be unique in loading a full CLP system before configuring PCI hardware.

Perhaps surprisingly, this imposes very few requirements on the OS. The SKB, like most of the components, executes in user space as in a classical microkernel design. However, CLP requires very little of the OS to be functional beyond basic (non-paged) virtual memory and a simple file system, initially from a RAM disk passed as a GRUB multiboot module at boot. The dynamic nature of the solution allows us to load further functionality after an initial PCI configuration when disks, networking interfaces, etc. come online.

Learning curve: Most OS programmers use C rather than Prolog to implement algorithms, and the learning curve for a language like Prolog is almost certainly steeper than for C. However, we feel someone with a basic knowledge of Prolog will find it easier to understand our code than a complex, imperative C version.

Furthermore, we are by no means the first people to try using logic programming in operating systems – for example, Prolog has been successfully used to provide network configuration logic in Windows [13].

7. Related work

This paper has considered a new approach to hardware programming, focusing on the specific problem of PCI resource configuration. The PCI specification [9, 23] describes the mechanisms and requirements for correct configuration of a PCI system, but does not specify any particular algorithm to be used in this process, leading to a variety of different (usually incomplete) solutions in current systems, as described in Section 2.7. These solutions are being iteratively refined and improved to handle more complex scenarios such as device hotplug [20, 29], leading to greater complexity.

A resource allocation algorithm for a hierarchical tree structure such as PCI has been patented by Dunham [10]. This algorithm sorts devices with fixed requirements according to their base address in ascending order, and all other devices according to their alignment requirements (size) in descending order. It then allocates resources to devices and bridges using a first-fit strategy starting at the lowest-level secondary bus, allowing it to determine the size requirement for the lowest-level bridge. Once its size is set, a bridge is then treated as a fixed-size device for allocation at the upper levels, and placed using the same first-fit allocation. Bridges are considered to have fixed address requirements if a device at any level below the bridge has a fixed requirement. As it encodes a specific traversal of the resource tree, this algorithm is roughly comparable to the postorder traversal discussed in Section 5.4 and used in varying forms by several current systems.

Rather than encode device configuration logic in low-level systems languages, we argue for wider use of declarative programming techniques. In this work, we specifically use constraint logic programming [15], a technique derived from logic programming and used to allocate resources in many fields. Prior applications of CLP include room allocation, task and job scheduling [2, 24], and indeed in our implementation we reused ECL^{i}PS^e primitives originally intended for task scheduling. Prolog has also been used in commercial systems such as Windows NT [13] to derive network configurations: a backtrack-based binding algorithm takes facts about interfaces of network modules and derives valid configurations, including the correct load order of modules, which it then stores to the registry. DEC developed a series of expert systems to ensure that selected component configurations that include CPUs and other hardware as well as software are valid and components are compatible to each other [6]. Hippodrome uses a solver to automatically configure minimal and still performant storage systems by analyzing workloads and iteratively searching a global minimum [1]. Declarative specifications of resources and resource requirements have also been used successfully by systems such as Condor [18, 28]. In the context of the Semantic Web, the resource description format (RDF) [30] is widely used to represent and reason online about resources. RDF is expressively almost equivalent to the logic programming approach we present here (ignoring the constraint and optimization extensions we employ), and might form the basis for a viable alternative to ECL^{i}PS^e.

Declarative language techniques have also been applied to operating systems, to date largely in the area of resource allocation. The Infokernel [3] was an early advocate in the OS arena of making a rich representation of system information available for online reasoning. Singularity [27] uses XML manifests to reason about the resources used by a device driver. These manifests may be analyzed at driver install time to checking for resource conflicts. They also ensure the correctness of a driver's interaction with the OS through contracts on message channels. The related system Helios [22] also uses manifests, to define preferred affinities of message channels to other processes, and thus guide the placement of processes to CPUs in a heterogeneous system. Similarly, the Hydra framework [31] uses a declarative approach to reason about available resources in a heterogeneous system consisting of host CPUs and programmable offload devices. Using an XML-based description language, the Hydra runtime selects suitable offload processors, thus achieving greater utilization of processor resources while reducing complexity for the programmer.

The complexity of hardware access can also be reduced through declarative approaches. Devil [19], an IDL for hardware programming, uses a declarative specification of device ports (base addresses), registers and their interpretation to generate low-level code for device access. This leads to simpler and more understandable code for device drivers, in an attempt to improve driver reliability. ATARE [16] uses a series of regular expressions to extract IRQ routing information from ACPI, without the need for the usual complex byte code interpreter.

Finally, SQCK [11] uses declarative queries to concisely implement a complete file-system consistency checker, which is also able to handle complex repairs. Together with the previous work, this signals what we see as a promising trend towards applying high-level declarative techniques to simplifying the construction of traditionally complex and error-prone systems software.

8. Conclusion and future work

In this paper we have investigated the case for applying declarative language techniques to low-level configuration of hardware in modern machines. In our initial experiments, we have shown that we can implement a solution to the complex PCI resource allocation problem using CLP with few lines of code, written in a natural and easy-to-evolve manner.

In addition, the approach provides considerable benefit from a clean division between policy and mechanism, and the further separation of general solution specification from the numerous special cases which inevitably occur when dealing with real software. However, care still must be taken when formulating the problem in CLP to avoid unacceptable explosions in execution time when searching for a solution.

The principle disadvantage of our approach is that it is heavyweight, in terms of memory footprint, execution time, and (when also considering the CLP runtime) total lines of code. Much of this is an artifact of our particular choice of a powerful, general purpose, but also mature (and therefore slower than the current state of the art) constraint logic programming system. While this choice has allowed us great freedom to explore the design space, a more appropriate solution for a product would compile the search algorithm into an efficient form when the OS was built, resulting in much faster execution and a smaller memory footprint.

Our view is therefore that the approach shows promise, and our experience in building an OS and delegating much hardware configuration functionality to the CLP engine has been positive so far. In our view, industry trends such as heterogeneous multicore, intelligent peripheral devices, sophisticated and reconfigurable interconnects, and partial cache coherence, combined with increasingly diverse platform configurations, strongly motivate a new and more systematic approach to reasoning about hardware configuration.

In our ongoing work, we are applying declarative techniques to other aspects of Barrelfish – in particular, more complete representations of the memory hierarchy than are available in typical OS NUMA support – and also applying logic programming techniques to naming and addressing the various processing elements in heterogeneous multicore systems.

Acknowledgments

We would like to thank our shepherd Michael Swift for his many helpful suggestions and the Barrelfish team at ETH Zurich and Microsoft Research, in particular Tim Harris, for numerous helpful suggestions for improving this paper.

References

[1] Anderson, E., Hobbs, M., Keeton, K., Spence, S., Uysal, M., and Veitch, A. Hippodrome: Running circles around storage administration. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2002), FAST '02, USENIX Association.

[2] Apt, K. R., and Wallace, M. G. *Constraint Logic Programming using ECL^{i}PS^e*. Cambridge University Press, 2007.

[3] Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., Burnett, N. C., Denehy, T. E., Engle, T. J., Gunawi, H. S., Nugent, J. A., and Popovici, F. I. Transforming policies into mechanisms with Infokernel. In *Proceedings of the 19th ACM Symposium on Operating System Principles* (Oct. 2003), pp. 90–105.

[4] Baldwin, J. H. Multiple passes of the FreeBSD device tree. In *BSDCan Conference* (May 2009). http://www.bsdcan.org/2009/schedule/attachments/83_article.pdf.

[5] Baldwin, J. H. About hot-plugging support in FreeBSD. http://www.mavetju.org/mail/view_message.php?list=freebsd-arch&id=3106757, Feb. 2010.

[6] Barker, V. E., O'Connor, D. E., Bachant, J., and Soloway, E. Expert systems for configuration at digital: Xcon and beyond. *Commun. ACM 32* (March 1989), 298–318.

[7] The Barrelfish Research Operating System. http://www.barrelfish.org/, December 2010.

[8] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating System Principles* (Oct. 2009).

[9] Budruk, R., Anderson, D., and Shanley, T. *PCI Express System Architecture*. Addison Wesley, 2004.

[10] Dunham, S. N. Method for allocating system resources in a hierarchical bus structure, July 1998. US patent 5,778,197.

[11] Gunawi, H. S., Rajimwale, A., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. SQCK: A declarative file system checker. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation* (Dec. 2008).

[12] Hewlett-Packard, Intel, Microsoft, Phoenix, Toshiba. *Advanced Configuration and Power Interface Specification, Rev. 4.0a*, Apr. 2010. http://www.acpi.info/.

[13] Hovel, D. Using Prolog in Windows NT network configuration. In *Proceedings of the Third Annual Conference on the Practical Applications of Prolog* (1995).

[14] The Importance of Implementing APIC-Based Interrupt Subsystems on Uniprocessor PCs. http://www.microsoft.com/whdc/archive/apic.mspx, January 2003.

[15] Jaffar, J., and Lassez, J.-L. Constraint logic programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (1987), pp. 111–119.

[16] Kauer, B. ATARE: ACPI tables and regular expressions. Tech. Rep. TUD-FI09-09, TU Dresden, Faculty of Computer Science, Dresden, Germany, Aug. 2009.

[17] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating System Principles* (Oct. 2009).

[18] Livny, M., Basney, J., Raman, R., and Tannenbaum, T. Mechanisms for high throughput computing. *SPEEDUP Journal 11*, 1 (June 1997).

[19] Mérillon, F., Réveillère, L., Consel, C., Marlet, R., and Muller, G. Devil: an IDL for hardware programming. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation* (2000), pp. 17–30.

[20] Microsoft. PCI multi-level rebalance in Windows Vista. http://www.microsoft.com/whdc/archive/multilevel-rebal.mspx, Nov. 2003.

[21] Microsoft. Firmware allocation of PCI device resources in Windows. http://www.microsoft.com/whdc/connect/PCI/pci-rsc.mspx, Oct. 2006.

[22] Nightingale, E. B., Hodson, O., McIlroy, R., Hawblitzel, C., and Hunt, G. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the 22nd ACM Symposium on Operating System Principles* (2009), pp. 221–234.

[23] PCI-SIG. *PCI Express Base 2.1 Specification*, Mar. 2009. http://www.pcisig.com/.

[24] Reis, L. P., and Oliveira, E. A constraint logic programming approach to examination scheduling. In *Proceedings of the 10th Irish Conference on Artificial Intelligence and Cognitive Science* (1999).

[25] Rusling, D. A. The Linux kernel. http://tldp.org/LDP/tlk/tlk.html, 1999.

[26] Schüpbach, A., Peter, S., Baumann, A., Roscoe, T., Barham, P., Harris, T., and Isaacs, R. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the 1st Workshop on Managed Multi-Core Systems* (June 2008).

[27] Spear, M. F., Roeder, T., Hodson, O., Hunt, G. C., and Levi, S. Solving the starting problem: device drivers as self-describing artifacts. In *Proceedings of the EuroSys Conference* (2006), pp. 45–57.

[28] Thain, D., Tannenbaum, T., and Livny, M. Distributed computing in practice: the Condor experience. *Concurrency: Practice and Experience 17*, 2–4 (2005), 323–356.

[29] TJ. PCI dynamic resource allocation management. http://tjworld.net/wiki/Linux/PCIDynamicResourceAllocationManagement, June 2008.

[30] W3C. Resource description framework, Feb. 2004. http://www.w3.org/RDF.

[31] Weinsberg, Y., Dolev, D., Anker, T., Ben-Yehuda, M., and Wyckoff, P. Tapping into the fountain of CPUs: on operating system support for programmable devices. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (2008), pp. 179–188.