

# Binary Translation & Virtualisation Challenges

## Introduction

---

The evolution of the computing landscape has three main drivers: (i) hardware architecture, (ii) software development and (iii) user requirements, as illustrated in Figure 1.

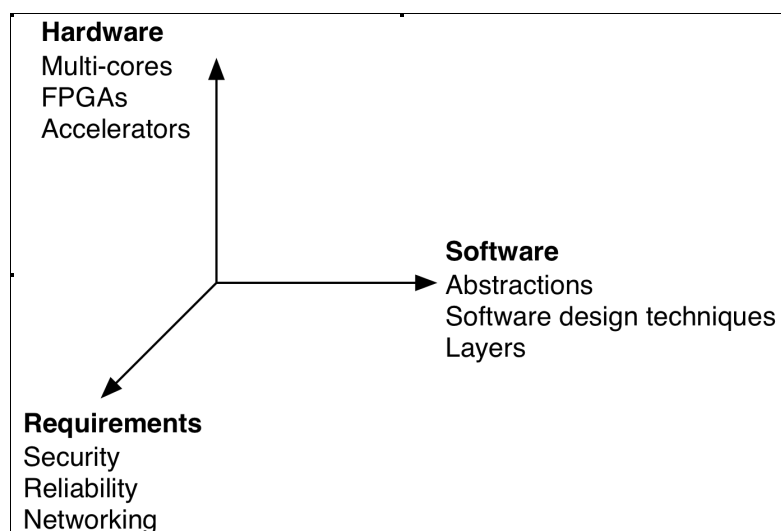


Figure 1: Three drivers of computing evolution

During the last couple of years, **computer hardware** evolution has shifted from increasing single core performance to increasing throughput by putting multiple closely coupled cores on one chip. Concurrently, the popularity of specialised accelerators keeps growing and reconfigurable hardware starts making headway as well.

A major difference with the past is that the **system architecture** is not fixed anymore, and that as a consequence software will have to be adapted to run well on the next generation heterogeneous multi-cores systems. The heterogeneous multi-core and many-core architectures present a multitude of optimisation opportunities, but exploiting them to the fullest extent requires a lot of effort and investments. Even more, fully exploiting the hardware to its limits might even become increasingly risky as evolutions in the fabrication process reduce the predictability and reliability of the end products.

**Software** complexity has meanwhile been evolving virtually in lockstep with the increasing architecture capabilities. Software layer upon software layer has been built to master the growing software complexity. Independently developed and maintained components plug into each of these layers to provide the necessary functionality and design flexibility. Additionally, the use of dynamic languages makes it increasingly difficult to statically predict the behaviour of a system, creating the need for more run-time optimisations.

At the same time, today's connected **business requirements** are more stringent than ever with strong security and reliability guarantees. Complex software systems are augmented with integrity checking, intrusion detection, obfuscation and encryption facilities to safeguard important personal and business data. These safeguards need to crosscut or be applied

separately to all layers and components, since an imperfection in just one place can compromise large sections of the entire system.

They all influence each other to some extent. New business requirements result in more complex software that requires faster hardware. Hardware improvements up the expectations of end-users regarding their software's performance and features. New software keeps pushing the latest hardware to the edge of its capabilities. On the other hand, the three drivers also evolve individually: neither the software nor the business requirements have explicitly asked for the transition to multi-cores; neither the hardware, not the business requirements have explicitly asked for managed languages; neither the hardware nor the software have asked for more stringent requirements w.r.t. security or reliability – actually they are struggling with it.

## Problem statement

---

The economic lifetime of software is many times longer than the economic lifetime of hardware. The real value (and also the complexity) of an information processing system lies in the specialised software, not in the generic hardware that is replaced at regular times anyhow. Therefore, companies want to protect their investments in software.

This evolution leads to the **hardware/software paradox**. In the past, hardware was considered to be complex and inflexible whereas software was simple and flexible. In spite of hardware's exponential performance evolution as illustrated by Moore's law, the software stack has in time overtaken hardware in terms of complexity and associated inflexibility. Therefore, rather than looking for software to run on their hardware platform, people are looking for hardware platforms to run their software on. Since platforms are evolving fast (and are expected to evolve even faster in the coming years with the advent of many-cores, accelerators and reconfigurable hardware), old platforms will become obsolete, requiring extra investments to port hard-to-change software to new hardware platforms.

**The big challenge is therefore to let the hardware evolve at its current rapid pace while keeping the software maintainable, stable and efficient.**

One family of techniques that can help with solving these challenges is binary translation and virtualisation (BTV). These techniques can assist with automatically transforming software stacks, ranging from dynamic optimisation and resource management to abstracting away the entire underlying hardware architecture or software layers. They can also help with providing greater insight into the software stacks. Furthermore, future co-designed hardware/software BTV platforms can offer additional advantages over the current approach of making the BTV layer, the other software layers and the programmer as oblivious to each other's existence as possible.

In what follows we will explain each challenge more in-depth and illustrate how BTV can help with addressing them. In case of the hardware level, we will also look at how hardware support can assist with BTV. Next we will zoom in on the associated BTV challenges, both for the near and medium term.

## Hardware

---

### BTV Advantages

---

#### Platform independence

---

Further increasing the sequential performance of superscalar processors is becoming steadfastly harder because it is getting increasingly difficult to extract instruction level parallelism. As a result, multiple lower performance processors, optionally in combination with high performance accelerators (reconfigurable or not) and interconnected with a Network-on-Chip, are introduced to better exploit the exponentially growing number of transistors than investing them all in one fat core could ever achieve. Such **heterogeneous multi-core processors** do not have a fixed architecture, but are rather characterized as a family of architectures (e.g. one x86 + two vector accelerators, or one PPU and 8 SPUs). The details of such an architecture are only known at load-time of the application.

BTV can help address the challenges associated with this evolution:

- A program compiled to a virtual intermediate format can be recompiled later for different architectural features, either statically or dynamically;
- A virtual intermediate format can express concepts not yet available in current hardware, thereby more easily enabling optimal compilation for future platforms;
- Whole-program optimisation, either by a link-time optimiser or by a dynamic compiler, can enable cross-ISA optimisation for heterogeneous multi-cores.

#### Adaptability to hardware variability

---

As the feature size further shrinks several reliability problems pop up: soft errors caused by alpha particles, process variability, lifetime reliability, hard faults. Fewer chips exit the fabrication process entirely fault-free or with the exact same specifications, and faults also develop more easily during the lifetime of the chips. This process variability is exacerbated by the increasing power density and associated thermal hotspots.

In some cases such defects can result in reduced performance, in others in faulty executions. BTV can deal with these issues using a number of techniques that are not available when using static approaches:

- Code and data layout can be adjusted per execution, enabling workarounds for stuck bits and similar issues in branch predictors, cache lines, memory, ...;
- Dynamic (re)compilation can generate alternate code to avoid using non-functional CPU instructions and registers;

#### Resource management

---

Operating environments have become increasingly dynamic, especially if several software applications are shared the same multi-core hardware. Software requirements can vary significantly from one moment to the next, as do the available resources. Power-efficiency dictates that as few components as possible should be in use at any given time, while thermal issues may conversely require distributing a workload over multiple machines or, at a smaller scale, functional units.

- Virtualised environments enable easy and efficient dynamic reallocation of resources on an as-needed basis for performance, thermal and efficiency reasons;
- Virtualised environments can hide the underlying hardware, thereby enabling transparent substitution of one kind of hardware for another if the expected kind happens to be unavailable (e.g., replacing a specialised accelerator with several generic cores emulating this accelerator).
- The use of certain components or functional units can be dynamically adjusted to protect them from overheating through so-called activity migration.

## BTV challenges

---

### Hardware support for BTV

---

Today, hardware support for BTV often means that a few extras have been bolted on an existing platform to address some of the worst performance problems. In other cases, new processors were designed to efficiently virtualise another architecture (e.g., the Transmeta Crusoe).

In order to fully harness the opportunities offered by BTV, hardware-software co-design is however indispensable. An example of this approach can be seen in Sun's Portmeiron project. Open research questions in this area include:

- Integrating support for legacy platforms with reasonable performance characteristics;
- Examining the impact on design effort and hardware cost;
- Investigating the cross-layer optimisations that such an approach enables.

### Dealing with hardware variability

---

Hardware variability comes in many forms:

- **Heterogeneous multi-cores** — In the embedded world, most systems consist of a combination of general-purpose cores and DSPs or accelerators. The general purpose cores themselves may be different implementations of the same ISA, or cores that feature different ISAs;
- **Heterogeneous clouds** — Server farms, also called computing clouds, may contain different kinds of multi-core processors. When processes are able to migrate live between different processors, this variability can be seen as variability over time;
- **Reconfigurable cores** — FPGAs and friends can be seen as processors that can vary over time. So can soft-cores that are implemented on top of FPGAs;
- **Process variability** — Process variability can also be seen as processor variability, as processors with different properties (clock frequency, power consumption, ...) will leave the fab. Furthermore, process variability can be seen as hardware variability over time, as defects will occur as a result of too much stress or wear.

In all of these cases, a programmer ideally should be able to write software in a processor-independent manner. This will ensure that the software can run on all potential target processors, and potentially that a running program can migrate between different cores.

Total virtualisation, including support from the hardware, is the paradigm that allows us to achieve this:

- A generic intermediate format that enables specifying the program at a level that allows for easy adaptation to varying hardware conditions;
- Meta-information providing information that enables taking advantage of different hardware features;
- The ability for the hardware to communicate faults, thermal properties and other variable characteristics to the BTV system so the latter can take these into account.

The research challenges related to realising these advantages can be categorised as follows:

- near term:
  - *resource management*
  - *whole-program optimization for heterogeneous multi-cores*
  - *speed up hw simulation – hybrid simulation*
  - HW support for btv

- virtualisation support in ISA
  - virtual interface between HW & SW (BOA) – virtual ISA?
  - multiple architectures in a single VM
  - virtualizing everything
- medium term:
  - *compensate hardware defects*
  - *virtualised hw accelerators*

## Software

---

### BTV advantages

---

#### Decoupling design and execution time, cross-layer optimization

---

Many software design techniques such as layering and component-based design have associated run time costs. This is due to the conceptual application design being reflected in the resulting binary code, such as various API layers calling each other, abstractions shielding implementation details, and application/OS boundaries.

BTV can assist in reducing this run time overhead by transforming the code so the design boundaries no longer exist as such at run time:

- Dynamic optimisation across software layers using dynamic binary translation and process virtual machines;
- Dynamic optimisation of the application/OS interaction at the system virtual machine level.

#### Tool chain flexibility

---

Tool chains from hardware vendors are often at least partially proprietary, causing problems for customers if they want to add functionality. This problem manifests itself particularly often in the embedded world. Similarly, a vendor tool chain may not support a particular programming language or dialect desired by a customer.

Solutions offered by BTV in this field are:

- Static compilation to an open intermediate format enables multiple tools (both open and proprietary) to easily work successively on the same program code;
- Depending on the choice of the intermediate format, it may also be possible to feed it to a Just-in-Time compiler for added flexibility;
- As soon as a frontend for compiling a source language into the intermediate format exists, no extra support from the vendor back-ends is required to support this language.

### BTV challenges

---

#### Meta-information about client programs

---

Many transformations and opportunities are only available when the appropriate information about the program to be translated or virtualised is available. This information can sometimes be automatically reverse-engineered or gathered using static analysis or run time instrumentation, but such techniques seldom are 100% reliable and complete.

In order to enable many of the opportunities presented in this document, we need a common format to specify all required information in a platform- and framework-independent way throughout the entire software stack running on top of the BTV platform. Examples of such information are:

- Phase change markers: the can help optimise resource allocation and improve dynamic optimisation;
- Interaction specifications: description of available hooks (for the BTV framework or other processes) that enable observing and influencing the behaviour of the process, e.g. pause, terminate, insert/remove actuators;
- Semantic information about the program: static and dynamic type information, memory allocation interface, run time type information description and location, ...
- Resource requirements: minimal QoS of NoC to deliver predictable, real-time performance

## Interacting with the hosts

---

Until now very little work has gone into specifically designing applications and programming models for running inside BTV environments. Binary translation is almost invariably a completely independent add-on technique used for dealing with unforeseen issues that somehow cannot be adequately dealt with using higher-level techniques.

Similarly, system virtualisation is mostly used for dealing with legacy systems or otherwise existing software stacks. Even languages specifically designed for process virtual machines offer few to no introspection or hooks into the VM, and are indeed designed so the programmer can be completely oblivious to the fact that he is working in a virtualised environment.

Similar to having a standard way of interacting with the client applications of a BTV stack as described in the previous point, a common standard for interacting with and interrogating the host side of the BTV equation would open up many opportunities. Paravirtualisation does this already to some extent, but is often very application- and framework-specific.

We need a more general approach, whereby a generic and extensible interface is defined that can be implemented by the various BTV systems. Any application running on top of such a BTV could then use this infrastructure for, e.g.,

- On-demand check pointing/rollback: an application can ask the VM to checkpoint its state at known “safe points”;
  - Execution obfuscation: an application can ask for obfuscating/diversifying parts or all of its execution (trading performance for less observability);
  - Environment introspection: does the execution environment offer certain guarantees (digital restrictions on output, signed code, license compatibility of all code);
  - Self-tuning: based on known available resources (energy, cores, memory, ...) tune up/down the quality of the computations/output.
- 
- near term:
    - *(dynamic) optimizations across software design layers*
    - virtualised ISAs
    - debugging
  - medium term:
    - transfer techniques used in enterprise environments to embedded systems

## Requirements

---

### BTV advantages

---

#### Security

---

Security cannot be guaranteed by only looking at individual components. Often, breaches occur due to unforeseen interactions or race conditions between different components. Another issue is that when a software component checks its peers, it is possible for the component itself to be compromised since it runs inside a potentially vulnerable system.

A more global and preferably external view of the system is required in order to observe and secure the interactions between various subsystems, which BTV can help with:

- A system virtual machine can monitor the integrity of the guest software, in particular the kernel;
- Virtualisation can be used to create trusted sandboxes which are isolated from the rest of the system, thereby reducing the potential harm caused by breaches;
- Dynamic instrumentation and compilation can be used to dynamically insert checks in foreign code;
- Randomising the virtualised environments can thwart many side-channel attacks.

#### High availability

---

Even a few seconds of downtime in a mission critical system can lead to tremendous losses. Redundant hardware can help in dealing with sudden failures, but the setup and maintenance of different machines, which all have to operate in a synchronised fashion, is a costly and time-consuming affair. Similarly, replacing a system with more powerful ones is a non-trivial affair if downtime is unacceptable.

Such issues can however be dealt with by virtualizing the operating environment:

- Seamless live migration between virtual machines running on different physical machines allows for transparently handing over the responsibilities of one machine to another, even in case long-running transactions are involved;
- The hardware dedicated to a single virtual machine can be expanded in a transparent way;
- Resource isolation can be used to ensure an adequate Quality of Service for all virtual machines running on a single host.

#### Vendor neutrality

---

Many hardware components are combined to form larger systems. If one such component is suddenly no longer available for some reason, this may cause serious problems for the system integrator. Similarly, if vendor support turns out to be subpar, being dependent on this vendor for tools and/or components is undesirable.

BTV however allows for hiding the actual underlying hardware:

- BTV can be used to emulate or support legacy components, thereby protecting R&D investments against supplier failures and the phasing out of older components;
- Basing development on an open virtualised intermediate format makes it more easy to switch the underlying hardware and increases the choice of available development tools;

#### Ubiquitous computing – code mobility

---

Large-scale computing clouds are springing up all over the world. The systems in such a massive collective are very likely to diverge in terms of hardware features over time, yet people should be able to submit a single program to run on it. Similarly, mobile devices are

increasingly becoming extensions of people's desktop machines, and transferring running programs from one device to another would be very useful.

BTV is one way of achieving this:

- A generic intermediate format can be recompiled on the fly for a different platform;
- A virtualised environment allows for more easily capturing a program's relevant state and transplanting it to a different device;

## BTV challenges

---

### Performance modelling

---

Modelling and understanding the behaviour of a program is hard. Modelling the behaviour of an entire BTV stack with different client programs running on top of them is even harder. Nevertheless, insight into the low level behaviour of such stacks, predicting performance consequences of software and hardware decisions and determining bottlenecks (preferably ahead of run time) is of paramount importance to tune and improve such environments.

Important aspects in this area are:

- Meta-information available in all layers of the system that enables a vertical view crosscutting the entire software stack;
- Modelling the interactions between various layers, cores, hardware variants, dynamic code changes and environment adaptations;

### Certification and validation

---

The complexity of BTV systems means that the validation required before they can be used in mission-critical situations is extremely hard. Nevertheless, such validation and subsequent certification is very important given the increasing ubiquity of BTV. A number of steps in this validation process could be:

- The specification of which program transformation can be validated and to what extent (defining invariants for different classes of transformations);
- Assessing existing validation techniques and their relevance when applied to BTV (different approaches will be required for generic intermediate code and legacy code);
- BTV on the other hand also enables new validation techniques, e.g., here as well meta-information could be used to specify constraints that can be checked at run time.
- near term:
  - *seamless live migration*
  - *security*
    - *monitoring guest kernel integrity*
    - *trusted platforms/sandboxes*
  - *portability (one toolchain to rule them all)*
  - *high availability*
  - *performance modelling*
  - *hard real-time*
- medium:
  - *ubiquitous computing*
  - *long term availability/support for legacy (Rolls-Royce)*
  - *VM certification*

## Who is who in BTV?

### Optimisation across software layers

**Who:** IBM, UGent, UC Santa Barbara, UC Boulder, Transitive

- Maebe, J., Buytaert, D., and Eeckhout, L. 2006. Javana: A System for Building Customized Java Program Analysis Tools. In Proceedings of OOPSLA 2006, pp. 153-168.
- Mysore, S., Mazloom, B., Agrawal, B., and Sherwood, T.. Understanding and Visualizing Full Systems with Data Flow Tomography, Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 2008. Seattle, WA.
- Hauswirth, M., Diwan, A., Sweeney, P. F., and Mozer, M. C. 2005. Automating vertical profiling. In Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (San Diego, CA, USA, October 16 - 20, 2005). OOPSLA '05. ACM, New York, NY, 281-296.
- Hu, S. and John, L. K. 2006. Impact of virtual execution environments on processor energy consumption and hardware adaptation. In Proceedings of the 2nd international Conference on Virtual Execution Environments (Ottawa, Ontario, Canada, June 14 - 16, 2006). VEE '06. ACM, New York, NY, 100-110.
- D. Chagnet, B. De Sutter, B. De Bus, L. Van Put, K. De Bosschere, Automated reduction of the memory footprint of the Linux kernel. ACM Transactions on Embedded Computing Systems (TECS). ACM Press. Vol. 6 (4). 2007. pp. 23

### Resource Management

**Who:** Georgia Tech, Rice U, Q-Layer, Sun

- Kumar, S. and Schwan, K. 2008. Netchannel: a VMM-level mechanism for continuous, transparent device access during VM migration. In Proceedings of the Fourth ACM SIGPLAN/SIGOPS international Conference on Virtual Execution Environments (Seattle, WA, USA, March 05 - 07, 2008). VEE '08. ACM, New York, NY, 31-40.
- Ongaro, D., Cox, A. L., and Rixner, S. 2008. Scheduling I/O in virtual machine monitors. In Proceedings of the Fourth ACM SIGPLAN/SIGOPS international Conference on Virtual Execution Environments (Seattle, WA, USA, March 05 - 07, 2008). VEE '08. ACM, New York, NY, 1-10.

### Performance modelling

**Who:** UGent, Intel, DaCapo team

- Apparao, P., Iyer, R., Zhang, X., Newell, D., and Adelmeyer, T. 2008. Characterization & analysis of a server consolidation benchmark. In Proceedings of the Fourth ACM SIGPLAN/SIGOPS international Conference on Virtual Execution Environments (Seattle, WA, USA, March 05 - 07, 2008). VEE '08. ACM, New York, NY, 21-30.
- Georges, A., Buytaert, D., and Eeckhout, L., Newell, D.. 2007. Statistically Rigorous Java Performance Evaluation. SIGPLAN Not. 42, 10 (Oct. 2007), 57-76.
- Andy Georges, Lieven Eeckhout and Dries Buytaert, Java Performance Evaluation through Rigorous Replay Compilation, OOPSLA 2008.
- S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffman, A. M. Khan, R. Bentzur A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century. Communications of the ACM, 2008. To appear.

### Guaranteeing secure execution

**Who:** U Toronto, HP Labs, ARM, Microsoft

- Anderson, M., Moffie, M., and Dalton, C. April 2007. Towards Trustworthy Virtualisation Environments: Xen Library OS Security Service Infrastructure. Technical Report. HP Trusted Systems Laboratory, Bristol

- Asrigo, K., Litty, L., and Lie, D. 2006. Using VMM-based sensors to monitor honeypots. In Proceedings of the 2nd international Conference on Virtual Execution Environments (Ottawa, Ontario, Canada, June 14 - 16, 2006). VEE '06. ACM, New York, NY, 13-23

**HW support for BTV**

**Who:** AMD, IBM (BOA), Intel, Transmeta, U of Virginia, Sun Microsystems (Portmeiron)

**Virtualising ISA's**

**Who:** IBM (DAISY), Transmeta, HP Labs, ST, VirtuaTech, Sun Microsystems (Portmeiron)

**Cross-ISA rewriting**

**Who:** HP Labs

- Chapman, M., Magenheimer, D., and Ranganathan, P. May 2007. MagiXen: Combining Binary Translation and Virtualization. Technical Report. HP Laboratories, Palo Alto.