Mutant Components: Efficiently Managing Multiple Implementations

João Gabriel Reis, Antônio Augusto Fröhlich.
Software/Hardware Integration Lab
Federal University of Santa Catarina
 Florianópolis, SC, Brazil
{jgreis,guto}@lisha.ufsc.br

Abstract—The development of embedded system applications is driven by the usage of models of computation and their interaction with the underlying programming language and the interfaces of system components. Behind each interface there is a series of software and hardware operations with different shapes according to the availability of implementations, hardware accelerators, or processor cores, and the environmental conditions faced by the embedded system such as the power consumption or even the chip’s temperature. For example, when running out of battery, a mobile device can decide to reconfigure the implementation of critical components for one that consumes less energy by delivering a functionality with a lower quality-of-service without depleting the battery. To cope with such adaptive systems without breaking the interface each component provide to the application, we propose a framework for mutant components whose implementation can be reconfigured during runtime. The system synthesis delivers a tailored wrapper for each component according to the number of implementations it has. We evaluate our proposal by profiling the execution time of methods from three components (AES, ADPCM, and DTMF) with a hardware and a software implementation to evaluate the overhead incurred to the resulting architecture regarding its the execution time.

Keywords—System-level design, HW/SW co-design, High-level synthesis.

I. INTRODUCTION

Rigid partitions of components or modules in a hardware/software co-design flow can lead to suboptimal choices in systems with dynamic or unpredictable runtime requirements. A partition can be seem as a combination of the micro architectural choices made to fulfill the application requirements and the substrate where it is deployed (e.g. multi-core CPU, hardware accelerator, a remote machine) captured by the component implementation. Each implementation has a unique footprint in terms of the quality of the operation provided which is result of the trade-offs made on design time and the substrate resources statically allocated for it. Changing the implementation currently deployed by a given component can help systems cope with dynamic non-functional requirements such as performance and power, hardware defects due to Negative-Bias Temperature Instability (NBTI) or Process, Voltage and Temperature (PVT) variations, or application requirements unforeseen at design time. A reconﬁgurable system could for example migrate the implementation of a multimedia encoder from a high quality software version running on a general-purpose core to a dedicated hardware accelerator or to an alternative lower quality version running on a low-power embedded microcontroller according to system load or power consumption.

Component interfaces isolate developers from architectural and implementation details, and even from specific operating systems and software environments. Behind each interface interaction, nevertheless, there is a series of software and hardware operations that may take different forms according to the availability of multiple implementations. Hardware/Software co-design techniques have expanded the implementation space even further: components can be partitioned and synthesized into different cuts of software and hardware components. The quality and timeliness of results from each implementation is dependent on the hardware environment as well as on explicit choices by the programmer (e.g. calling a double precision versus a single precision mathematical function). Finally, the cost — be it in processor cycles, energy, or hardware area — of performing a certain operation may also change according to system load as well as environmental and manufacturing-related variations (e.g. elevated power dissipation at higher temperatures). Given the programming model of applications and the richness of choice in component implementations, we can think of the embedded computing substrate not as the combination of processor, peripherals, field-programmable devices and other hardware, but instead, as the components and interfaces themselves. While there are choices in implementation and variation in quality and costs for each substrate, choices are typically static (decided at design time) and variations are typically not quantified and exploited, but simply suffered.

Binding a component interface to a single implementation that might not be optimal for the entire application life span (or even its majority) compromises the component versatility. Nevertheless, the application developer should not be overwhelmed by the numerous implementations that the same component might have. In most cases it is not be feasible to predict all the environmental conditions the application will face during deployment or to monitor them to chose the best implementation in the application itself. Instead, it is preferable to delegate those tasks to alternate software layers that posses a holistic understanding of the system [1] and open the possibility to transparently reconﬁgure a component implementation during runtime while keeping the same interface.

This work present the infrastructure to deploy components with an implementation that can be reconﬁgured during runtime which, from now on will be called a Mutant components. It differs from previous approaches in its syntax and semantics.
which are preserved for all implementations on all substrates a mutant component might possess. To evaluate the proposal, we profiled the executing time of methods from three mutant components (AES, ADPCM, and DTMF). All components have a software and a hardware implementation and were deployed in the EPOS framework [2] under the Xilinx Zynq-7000 platform using an ARM Cortex-A9 processor. When compared with their equivalent monomorphic versions (single implementation without the proposed infrastructure), for most cases the mutant component presents a small relative time overhead on method invocation.

II. MUTANT COMPONENTS

In this section, we introduce the guidelines used to design mutant components. Each component and its interface is mutant in terms of its implementation as well as its resulting quality of service. Mutant components allow for dynamic change between multiple software and hardware implementation choices, as well as adjustments in operation parameters for each of the choices. The resulting quality of service and cost for each component is dependent on the specific choice of implementation as well as the physical state of the system at any point in time.

A. Application-Driven Embedded System Design

For developing hardware and software implementations we rely on Application-Driven Embedded System Design (ADESD) which is a multi-paradigm domain engineering methodology to decompose a problem domain into reusable and scenario independent abstractions while capturing scenario related variations into a set of aspects [2]. Despite being initially proposed as a software engineering methodology used in the context of dedicated operating systems, ADESD concepts has been widely applied in the field of embedded systems development [3].

ADESD was also used to explore common implementation and communication interface for components deployed in the software and hardware domains [4]. This approach, called Hybrid Components, allows freely migrating the component from one domain to another in any phase of the system design process without needing to adjust the system itself. Based on Hybrid Components’s Aspect Oriented Programming (AOP) and Object Oriented Programming (OOP) techniques, developing unified descriptions of hardware and software components was also studied in ADESD [5]. Components designed following such principles are susceptible to both software and hardware synthesis using standard compilers and High-Level Synthesis (HLS) tools. This is possible through the isolation of specific hardware and software characteristics (resource allocation and communication interface) into aspect programs which are weaved with the unified descriptions only during the synthesis process. Therefore, the extraction of hardware/software implementation from the unified implementation happens directly through language-level transformations, thus facilitating compatibility with different C++-based HLS tools and design flows. By using ADESD concepts we provide an approach for deploying mutant components employing static metaprogramming techniques and a transparent interface from the application developer point of view.

B. Usage model

Multiple implementations of a mutant component are exposed to applications through a single, unified interface which in our approach is called inflated interface [2]. Generic, metaprogrammed software artifacts are used to handle differing function signatures, communication with hardware IP cores, and state migration between implementations. While all software implementations and IP cores bitstreams are included in the system image (typically kept in RAM), hardware accelerators are instantiated and freed on-demand, and therefore do not take up system resources, e.g. Field-Programmable Gate Array (FPGA) area, when not in use. Per-application constraints are used to guide the reconfiguration process and dictate which version of a component should be used under different circumstances. From the application’s standpoint, a call to a double precision exponential function is therefore identical to a call activating a hardware IP to calculate it or a remote call to a cloud resource.

Our approach assumes that applications dynamically instantiate (and destroy) the components they need, and the operating system automatically uses the component implementation that best suits its current needs. Multiple component implementations are available to applications through a single interface. For the application, a call to a software function is, hence, identical to employing a hardware IP core or a remote call to a cloud resource. The application programmer selects its preferred implementation of a given component for each application but, to cope with system load as well as environmental and manufacturing-related variations, the deployed implementation might change during runtime. To infer which implementation best suits the current system needs, the operating system opportunistically and speculatively monitors system load, hardware resource usage, and hardware performance parameters.

We divided implementations in three groups: software, hardware, and remote. Software implementations run in soft and hard core processors where the operating system is also being executed. Such implementations might use not only the CPU ALU but also tightly-coupled coprocessors such as floating-point units and Single Instruction, Multiple Data (SIMD) engines through special instruction sets. Hardware implementations can vary in microarchitecture details that result in different balances between power consumption, performance and resource usage. They can be implemented as dedicated circuits in Application-Specific Integrated Circuits (ASICs) or as reconfigurable modules in FPGAs. In both cases, the CPU executing the operating system interacts with them through I/O mechanisms such as Network on Chip (NoC) communication, Direct Memory Access (DMA) or memory-mapped I/O. The last group, remote implementations, comprises software and hardware implementations that are not being deployed on the same machine as the operating system. The implementation is accessed through a network device, which will forward the call to a remote machine and fetch the results.

We illustrate the usage of mutant components using the C++ programming language but, the same concepts could be applied to any programming language supporting object-orientation and static metaprogramming techniques. C++ classes abstract components, each component has a parametrized class whose static constant members describe the properties (traits) of a certain type. The only additional information the user must provide to the operating system is which component implementations suit best each application in order of preference through
the component’s traits. In Figure 1, the parameterized class
\texttt{Traits<Component>} denotes the traits of Component
for a given application in which the implementation order of
preference is Implementation\_2, Implementation\_0, Implementation\_1. With this information, the system can
adapt to process and environmental variations during runtime
by using implementations that best suit system’s requirements
while trying to utilize the user’s preferred implementations.

\begin{verbatim}
template<> struct Traits<Component>
{
   typedef LIST<Implementation\_2, Implementation\_0, Implementation\_1> IMPS;
};
\end{verbatim}

Figure 1: Example of the parametrized traits class for a generic
component.

Template metaprogramming techniques enable the user to
assign a priority to each component implementation in each
application without inferring runtime overhead using the traits
mechanism. \texttt{LIST} type is a metaprogrammed list of types
populated by the user containing the preferred implementations
of Component. Through C\++ template metaprogramming, it is
possible to obtain the number of types in the list and also a
type stored under a given index during compilation.

\subsection{C. Decoupling interface and implementation}

Independent of the implementation deployed in a given
moment in time, the component must provide a uniform
interface to its clients. Also, changing the implementation
behind a given component must be a transparent operation
and a clean interface must be provided to the entity in
charge of performing component reconfiguration. To comply
with such requirements, this work proposes an indirection
between the component interface and the implementation being
currently deployed. Moreover, any infrastructure deployed for
this purpose must not incur any overhead when the component
has a single implementation. Polymorphism was employed to
guarantee that the infrastructure remains scalable independent
of the number of implementations and static metaprogramming
techniques assure that polymorphism is deployed only when
strictly necessary.

Figure 2a presents the building blocks of the infrastructure
proposed for mutant components. A parametrized class named
\texttt{Handle} is introduced between the mutant component interface
provided to its clients and the component implementations. It
defines a first component wrapper whose main purpose is
to ensure the usage of a proper allocator for components,
independently of how they are statically or dynamically declared
by the client. \texttt{Handle} realizes the interface of the component
passed as a parameter and forwards invocations of its methods
to the implementation deployed in a given moment. It can point
to any user-selected implementation while still maintaining
references to other implementations for later usage. It also
provides the \texttt{reconfigure()} method as means of changing
its component current implementation during runtime. Hence,
mutant components are manipulated through their \texttt{Handles}.
Figure 2a depicts the component interface using a \texttt{Handle}
that aggregates up to \texttt{N} implementations. \texttt{Handle} keeps a
reference to each deployed implementation by means of a

\begin{verbatim}
Implementation\_0
operation()
Component
«Mutant»
Implementation\_N
operation()
Component
N
(a) Component binding to a \texttt{Handle} with \texttt{N} implementations.

Implementation\_0
operation()
Component
ʂ«Mutant»
Implementation\_N
operation()
Component
false
Client

(b) Component binding to a \texttt{Handle} with a single implementation.
\end{verbatim}

Figure 2: Mutant component framework.

\texttt{Implementation} pointers array, \texttt{_implementations} as
presented in Figure 3. \texttt{IF} is a metaprogram that returns its
second parameter if its first parameter evaluates as \texttt{true}
else, it returns the third one. Moreover, \texttt{Polymorphic} is
a metaprogram that defines a boolean after searching the list
of implementations for any implementation of a type different
then that at the list’s head. The \texttt{reconfigure()} method
can be easily implemented by changing the element to which
\texttt{_current} points to.

Notice that \texttt{Implementation} shares the same base class
\texttt{Component} as each of the component’s implementations. This
allows \texttt{Implementation} to point to any implementation
and invoke their methods. If a component has more than one
implementation, \texttt{Component}’s methods will be pure virtual
to comply with the fact that \texttt{Implementation} might point to
multiple implementations. Otherwise, they will only be defined
and implemented in the component’s sole implementation
with the \texttt{Implementation} inheriting from it resulting in
the architecture presented in Figure 2b. The differentiation
is necessary to remove the virtual function call overhead for
a single implementation. The \texttt{Component} behaviour can be
achieved with template specialization techniques as presented
in Figure 4.

A minimal implementation of \texttt{Handle}’s constructor is
shown in Figure 5. The template parameter \texttt{mutant} is set
to \texttt{true} if the number of implementations is bigger than
1 by checking \texttt{The list’s Length}. \texttt{Handle}’s constructor
is able to initialize each implementation and insert them in
implementations by employing template recursion as a looping construct [6]. The first invocation of helper() will initialize the first metaprogrammed list element defined in the component’s traits as previously exemplified in Figure 1 and keep a reference for it in the implementation pointers array. helper() will initialize each element until it reaches the last element in the list, with an index given by Traits<Common>::::IMPS::Length. _current is a pointer to the current implementation and can be used by the Handle to dispatch invocations to the component’s methods.

C++ namespaces are used to separate component implementation and the decoupling artifacts from its interface exported to the application. The former is defined in the System namespace while the latter goes on the Application namespace. The client does not know it is interacting directly with a Handle as in the application namespace the component is bind to its Handle already specialized with namespace Application (typedef System::Handle<System::Component> Component).

Figure 3: Handle method forwarding implementation reconfiguration.

Figure 4: Possible Component implementation.

D. Cross-domain interaction

Each implementation encapsulates the communication mechanisms necessary to exchange data and perform the computations on its substrate. In the software domain, components are objects which communicate using method invocation, while in the hardware domain, components communicate using I/O signals and specific handshaking protocols. For communication across different domains, the operating system must provide appropriate abstractions for hardware components and mechanisms for interrupt handling while the hardware must be aware that it is requesting a software operation. For instance, if given implementation uses a hardware IP to perform a calculation it is responsible for all mechanisms that must be deployed to exchange data between itself and the IP. The same applies for an implementation that depends on resources from different nodes of a network; it must operate the network stack to communicate with it.

To abstract communication patterns between components in different substrates, we employ an approach based on Remote Method Invocation (RMI) concepts from distributed object platforms [7]. Figure 6 illustrates an interaction between a component in the software domain with another in the hardware domain. Callee components are represented in the domain of callers by proxies. Channels deploy serializers and deserializers to marshal data structures exchanged by different domains. When an operation is invoked on a component’s proxy, the arguments supplied are marshaled into a request message and sent through a communication channel to the corresponding component’s agent. An agent receives requests, unpacks the arguments and performs local method invocations. The whole process is then repeated in the opposite direction, producing reply messages that carry eventual return arguments back.
The approach based on channels, proxies and agents that we have adopted for communication does not impose a single communication architecture. It can be used with different networking technologies, such as a shared bus, a NoC, or a DMA engine. For instance, when a bus-based communication channel is used, a proxy in hardware may be implemented as a memory-mapped slave device that notifies the CPU through interrupt requests when it has a message ready to be read. Alternatively, on a NoC based implementation, a packet-oriented interface would be used to transmit request messages. Such variability is related to choices regarding the hardware/software architecture of the chosen implementation platform and should not affect the system components. Finally, the resulting communication bandwidth is limited mostly by the chosen underlying communication technology.

E. State handling

To keep the system sane when changing a component implementation, the old implementation must provide a snapshot of its state so that it can be restored into the new one. By employing ADESD techniques for developing hardware and software implementations from a unified description, which in our case is written in C++, the state representation and the interface for handling it are uniform avoiding complex state migration issues. For simplicity, we consider that the high level description of each component implementation (hardware or software) provides a data structure that captures its current state as a set of variables and each of its implementations provides methods to save and to restore its state using such data structure. The methods, \texttt{save\_state()} and \texttt{restore\_state()}, are invoked by the entity performing the reconfiguration which is responsible for transferring the state from one implementation to another as one atomic operation. Figure 7 shows both methods used to manage the implementations state for a component whose current state comprises two variables, \texttt{\_a} and \texttt{\_b}. When synthesized into hardware, the state machine controlling its operation will be able to interpret its registers as the variables in the higher level description and return them when \texttt{save\_state()} is invoked by RMI. The same holds for \texttt{restore\_state()} that will map the data received by RMI into state machine registers.

Techniques for migrating and changing the numerical representation of the current state from one implementation to another are a complex matter and are outside this paper’s scope. For instance, there is no common convention for conversion between most numerical data types (e.g. how to convert a IEEE 754 binary64 number to a fixed point representation). Moreover, it is also not trivial to infer the type of each data value that constitutes the component state without embedding meta data in each value. Finally, the conversion must be performed only once (in the previous or in the next implementation) and a synchronization mechanism or convention must be employed to avoid multiple conversions.

III. RESULTS

To characterize the infrastructure overhead in terms of execution time we performed an experiment with three components: AES, ADPCM, and DTMF. AES implements the Advanced Encryption Standard (AES) ciphering and inverse ciphering. ADPCM is an Adaptive Differential Pulse-Code Modulation codec commonly used in telephony networks that compresses 16 bit audio samples into 1 bit. Finally, DTMF is a Dual-Tone Multi-Frequency (DTMF) detector that uses the Goertzel algorithm to verify if a set of audio samples contains the set of frequencies that define to a tone. The ADPCM is a stateless component while the AES, and the DTMF comprise the encryption key and the tone samples buffer as their state respectively. Despite the simplicity of the studied components, complex algorithms with multiple implementations can encapsulated as mutant components as long as each implementation has the same interface and implement the state handling functions. The reconfiguration process of the studied components including when it is deployed, its overhead, and latency were explored in a previous work [8].

Following ADESD techniques, a software and a hardware implementation for each component were generated from the same unified C++ description [5]. In the software domain, the hardware implementation is represented by its proxy that dispatches method invocations to the agent deployed in hardware which is responsible for invoking the component functionality. Trade-offs regarding execution time and hardware resources usage of the three components were previously explored [5], [9]. Despite being generally faster and consuming less energy than software implementations, hardware implementations depend on finite FPGA resources which might be already allocated for other components. In such cases cases the policy for

```cpp
class Implementation_0 {
public:
    struct State {
        int a;
        float b;
    };

    public:
    State save_state() {
        State state;
        state.a = a;
        state.b = b;
        return state;
    }

    void restore_state(State &state) {
        a = state.a;
        b = state.b;
    }

    private:
    int a;
    float b;
};
```

Figure 6: Cross-domain communication using proxies and agents.

![Diagram](image_url)
reconfiguration can be to push the implementations to hardware whenever there are available resources.

For the experimental analysis and evaluation we used a platform called EPOSSoC which presented in Figure 8. The CPU nodes execute software implementations of components and the operating system that orchestrates hardware implementations deployment. Software implementations run on the CPU node: A dual-core Cortex-A9 processor coupled with several peripherals executing the EPOS operating system. Rec nodes are reconfigurable partitions that might contain hardware implementations of components interconnected using a Real-Time Star Network-on-Chip (RTSNoC) [10] based scheme. A bus-based interconnect was discarded as it is not the most suitable choice for more heterogeneous designs in which hardware implementations play active roles [11]. The RTSNoC consists of routers with a star topology that can be arranged to form a two dimensions mesh. Each router has eight bidirectional channels that can be connected to cores or channels of other routers. Moreover, the rec nodes are connected to RTSNoC routers ports and one of the RTSNoC routers ports is connected to the CPU node through an AMBA bridge.

The hardware platform was prototyped in ZedBoard, a development kit based on Xilinx’s Zynq-7000, a System-on-a-Chip (SoC) couples a reconfigurable fabric with an ARM Cortex-A9 dual core CPU with several peripherals integrated in a single die. Xilinx’s Vivado HLS 14.2 was used to obtain RTL descriptions of the hardware implementations described using ADESDF. The software was compiled with GCC 4.4.4 targeting the ARMv7 ISA using level 2 optimization enabled by GCC’s –O2 flag while the hardware was prototyped using Xilinx’s Vivado 14.2 for RTL hardware synthesis. Synthesis constraints were adjusted on Vivado to minimize circuit area considering a maximum frequency of operation of 100 MHz. The CPU clock frequency is 666.666 MHz while hardware implementations clock frequency is 100 MHz. Table I gathers the FPGA resources used by each element in the EPOSSoC. The resource utilization of each component’s hardware implementation comprises the component functionality and the infrastructure necessary to communicate with software comprised in its agent. The estimated area is the arithmetic mean of the amount of each particular resource, weighted by its total amount available on the target device.

Table I: Hardware resources utilisation for the EPOSSoC in the XC7Z020 SoC.

<table>
<thead>
<tr>
<th></th>
<th>Flip-flops</th>
<th>LUTs</th>
<th>Memory</th>
<th>Estimated area (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTSNoC router</td>
<td>562</td>
<td>855</td>
<td>0</td>
<td>0.70</td>
</tr>
<tr>
<td>AES</td>
<td>1576</td>
<td>1595</td>
<td>6</td>
<td>2.92</td>
</tr>
<tr>
<td>ADPCM</td>
<td>519</td>
<td>642</td>
<td>1</td>
<td>0.80</td>
</tr>
<tr>
<td>DTMF</td>
<td>646</td>
<td>368</td>
<td>1</td>
<td>0.66</td>
</tr>
<tr>
<td>FPGA (XC7Z020)</td>
<td>106 400</td>
<td>53 200</td>
<td>140</td>
<td>100.00</td>
</tr>
</tbody>
</table>

All methods profiled were repeated 10 000 times and the total execution time was divided by 10 000 to calculate the average execution time. The time measurements were made using an oscilloscope and a GPIO pin to signal when the operation started and ended. The execution time of both versions of the AES performing the encrypt() and decrypt() methods were profiled. encrypt() performs the direct AES cipher on a 128 bit block stored on 16 B while decrypt() performs the inverse operation. As for the ADPCM, both encode() and decode() methods used to compress and decompress a 16 bit audio sample respectively were profiled. Finally, DTMF’s detect(), the method responsible for analyzing if a tone is found in a buffer with 700 samples and returning it, was profiled. The results are presented in Figure 9 and account for the method execution time for a mutant component with two, and one implementations and the component with a monomorphic implementation.

Despite the virtual method call in mutant components with two implementations, the total execution in some cases does not suffer a noticeable increase as the monomorphic method execution time is much bigger that the virtual function call overhead. Nevertheless, for methods with smaller execution times, the overhead of being used by a mutant component with two implementations translates in an relatively large increase in the total execution time. For instance, the software implementation of ADPCM::decode() which is a fast compared to others evaluated (0.86 μs) suffers an increase of 22% on its execution time compared to the method execution on a monomorphic component. On the other hand, a slower method as the hardware implementation of AES::decrypt() (4008.00 μs) has a barely noticeable increase when deployed on a mutant component with two implementation.

Notice that the software implementation of AES::decrypt() when deployed on a mutant component with a single implementation is slower than on a mutant component with two implementations. Despite being visible in the assembly code generated by the compiler that a virtual function call is performed on the latter case implying that a larger number of instructions is executed, we attribute...
The anomaly to a possible latency hiding mechanism in the Cortex-A9 CPU which employs an out-of-order speculative issue superscalar execution 8-stage pipeline.

IV. RELATED WORK

Several efforts focus on developing computing environments that can help developers to leverage reconfigurable resources and simplify its development flow. Such computing environments provide abstractions and interfaces to application programmers familiar with systems software development.

ReconOS [12] provides a common abstraction layer for software and hardware threads as well as a set of communication and synchronisation primitives for them. Low-level synchronisation and communication of hardware threads are managed by theirs Operating System Interface, an artifact embedded in hardware threads. It focuses on real-time applications and multithreading providing to the user an Application Program Interface (API) based on the eCos operating system as well as on Linux.

A higher-level approach to reconfigurable computing is proposed by Abel [13] that advocates describing reconfigurable modules in a Java-like language called Parallel Object Language. Though partial reconfiguration techniques, reconfigurable modules can be created and destroyed at runtime just like regular software objects. The modules can be tested in verified using a Java Emulator and then translated to VHDL for synthesis. Communication architecture is centered around FIFOs and multiplexers for parallel data exchange controlled by a software scheduler. The scheduler also that orchestrates reconfigurable modules loading into partially reconfigurable regions.

Moreover, Cemin et al. propose a framework for multi-agent systems that supports hardware/software migration among different nodes of a distributed system [14]. Their framework allows the dynamic reconfiguration of distributed application to cope with variable system requirements. The communication between agents is handled transparently by proxies that handle all the low-level intricacies of hardware communication. Their approach is implemented on top of the Jade framework, a Java-based multi-agent systems framework.

One broad approach for coping with unpredictable runtime variations in performance, energy, and user behavior is to have multiple code paths available to perform certain critical functions. Each of these paths may be better suited for a given system state or user input. Petabricks [15] and Eon [16], for example, feature language extensions that allow programmers to provide alternate code paths. In Green [17], a combination of a calibration phase and runtime accuracy sampling are used by the application to define which function to execute from a set of possible candidates. In Eon and Levels [18] the runtime system dynamically chooses paths based on energy availability. Compared to traditional algorithmic choice research, and in particular to the ViRUS framework [19] that shares our motivations for dynamic handling of variability events, reconfigurable computing adds a dimension of hardware reconfiguration, whereby one of the code paths may be offloaded to specialized hardware.

The code used to implement a given API function may also be changed at the binary or bytecode level. Dynamic recompilation techniques [20] test different optimization techniques at runtime, so that code is matched to the capabilities of the hardware which is running it. Dynamic recompilation may be performed in a system-driven manner, with minimal support from applications, providing the same adaptation knobs as in compile time optimization, e.g., loop unrolling, memory optimization, and automatic parallelization. Our work resembles dynamic recompilation in that the multiple code paths are explored automatically by the runtime system without application intervention, but it is closer to algorithmic choice in that the alternative implementations are defined at design time. In a similar fashion to dynamic recompilation, reconfigurable computing can be used to explore multiple code paths automatically by the runtime system without application intervention if alternative implementations are defined at design time just like with algorithmic choice.

Application and hardware parameters can also be dynamically adapted to explore energy, quality, and performance.
trade-offs [21]. Dynamic voltage and frequency scaling is the canonical example for hardware tuning. In software, Green [17] provides an adaptation modality where the programmer provides “breakable” loops and a function to evaluate quality of service for a given number of iterations. The system uses a calibration phase to make approximation decisions based on the quality of service requirements specified by the programmer. At runtime, the system periodically monitors quality of service and adapts the approximation decisions as needed. In the mobile context, Powerleash [22] is used to adapt the work of background tasks according to a desired rate of energy consumption. Adaptable software parameters can be used to maximally leverage the underlying hardware platform in presence of variations, for example in multimedia applications [23]. Finally, self-awareness can be extended to SoCs platforms coupled with cross-layer sensing and actuation [1]. The general mechanism of observe-and-adapt from parametric control can be used in reconfigurable systems to add a dimension of choice in software and hardware components that can be dynamically chosen to implement a given functionality.

The infrastructure for mutant components we propose can be seen as complement to most previously discussed works. We acknowledge that all those efforts present solutions for managing different implementations for system component using software and/or hardware techniques. Nevertheless, in most cases it is the applications programmer’s role to juggle with multiple implementations using different interfaces specially when they are deployed in different substrates, e.g. FPGAs and CPUs. In fact, our work could be combined with most of them to transparently build a reconfigurable system with components with a unified interface.

V. CONCLUSION

In this work we proposed a framework for mutant components whose implementation can be reconfigured during runtime while keeping the same interface. Mutant components deliver to applications interfaces that may be realized through many different implementations. While the syntax and semantics of the interface is preserved across the different versions, the system may at any time pick any of the versions that is better suited for the current execution context. Three mutant component were evaluated: AES, ADPCM, and DTMF. Results show that when compared with their equivalent monomorphic versions, for most cases the mutant component presents a small relative time overhead on method invocation.

This work focus on providing the infrastructure necessary for transparently deploying runtime reconfiguration for computing systems willing to exploit the variations in their environment and adapt to unpredictable application requirements. We are currently working on general guidelines to define when to reconfigure the system or even which is the best reaction for a particular set of conditions based on machine learning techniques exploring Power Management Unit (PMU) data.

REFERENCES