X-Ware: Mutant Computing Substrates

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

Lucas F. Wanner
Institute of Computing
University of Campinas, Campinas, Brazil
Email: lucas@ic.unicamp.br

***Abstract*—In this paper we introduce X-Ware, a framework for computing whereby components used by an application can take different forms and characteristics across the lifetime of the system in order to adjust to dynamic application requirements. In particular, we explore two aspects of system mutability: dynamic *choice* of implementations for certain system components (e.g., leading to different trade-offs between quality and resource usage); and the change in non-functional characteristics of these components (e.g., speed and energy consumption) due to process and environmental variations. We demonstrate X-Ware with a library of mathematical APIs that can help components save up to 93% in energy and 94% in execution time by tolerating a small degradation in quality.**

I. Introduction

Modern embedded applications are programmed arguably not through a series of hardware instructions, but through the specification of models and their interaction with language and API constructs. APIs — *Application Program Interfaces* — isolate developers from architectural and implementation details, and even from specific operating systems and software environments: a WebSocket `send` call performs the same function on Internet Explorer running on a Windows PC as in does on the iPhone's Safari Browser. Behind each API call, nevertheless, there is a series of software and hardware operations that may take different forms, for example, according to the availability of multiple implementations, hardware accelerators, network interfaces, or processor cores. Hardware/Software co-design techniques have expanded the implementation space for APIs even further: models and APIs can be partitioned and synthesized into different cuts of software and hardware components.

The *quality* and *timeliness* of results from each interaction with an API function is dependent on the specific implementation and 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 implementing components and APIs, 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 APIs themselves. While there are choices in implementation and variation in quality and costs in this substrate, choices are typically static (decided at design time) and variations are typically not quantified and exploited, but simply suffered.

A component implementing a mathematical exponentiation function, for example, could be realized in software in double or single precision, or through a dynamically instantiable hardware accelerator IP. The resulting performance and quality varies across each of the implementations. The energy cost of performing the exponential operation varies not only across the multiple implementations, but also across the lifetime of the system due to hardware and environmental variations. An application may *prefer* to use the software-based double precision version; it may *tolerate* the use of the single precision version when the system is overloaded; and it may *require* the hardware-accelerated version for certain real-time operations.

In this paper, we introduce X-Ware, a novel framework for *mutant* computing substrates. Each component and API in X-Ware is mutable 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.

Multiple implementations of a given component in X-Ware are exposed to applications through a single, unified API (which in our approach is called *inflated interface*). From the application's standpoint, a call to a double precision exponential function is therefore identical to a call activating an exponentiator hardware IP or a remote call to a cloud resource. Generic, metaprogrammed adaptors are used to handle differing function signatures, communication with hardware IPs, and state migration between implementations. While all software implementations and IPs 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 (particularly FPGA area) when not in use. Per-application constraints are used to guide the mutation process and dictate which version of a component should be used under different circumstances.

X-Ware differs from previous approaches in its syntax and semantics for hybrid components which are preserved across the multiple substrates and the reconfiguration process which is divided into multiple steps. We demonstrate X-Ware with a library of mutant mathematical components implemented for the EPOS RTOS framework [1] under the Xilinx Zynq-7000 platform using an ARM Cortex-A9 processor. When compared with their equivalent (highest quality) monomorphic API implementations, X-Ware components used up to 93% less energy and its execution time decreased 94% with a small quality degradation. Monitoring of relevant system parameters, dynamic hardware instantiation, and component mutation take negligible time in X-Ware, and fit comfortably into the RTOS's slack time.

II. Related Work

Our work is related to three major threads of research: algorithmic choice, parametric software and hardware control, and dynamic hardware reconfiguration.

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, for example. Petabricks [2] and Eon [3], for example, feature language extensions that allow programmers to provide alternate code paths. In Green [4], 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 [5] the runtime system dynamically chooses paths based on energy availability. Compared to traditional algorithmic choice research, and in particular to the ViRUS framework [6] that shares our motivations for dynamic handling of variability events, our work 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 [7] 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.

Application and hardware parameters can also be dynamically adapted to explore energy, quality, and performance trade-offs [8]. Dynamic voltage and frequency scaling is the canonical example for hardware tuning. In software, Green [4] 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 [9] is used to adapt the work of background tasks according to a desired rate of energy consumption. Finally, adaptable software parameters can be used to maximally leverage the underlying hardware platform in presence of variations, for example in multimedia applications [10]. Our work shares the general mechanism of observe-and-adapt from parametric control, but adds a dimension of choice in software and hardware components that can be dynamically chosen to implement a given functionality.

There is a large body of work in dynamic hardware reconfiguration [11], and particularly in leveraging dynamically reconfigurable resources in modern FPGA platforms [12], [13]. Runtime systems such as BORPH [14], FUSE [15] and ReconOS [16] provide drivers and OS-level support (e.g., file interfaces) for reconfigurable resources, but typically leave the management of migration up to application software. Closer to our work is the SPREAD programming model [17] whereby threads can migrate to specialized hardware resources and back to general purpose processors at runtime.

Our work differs from previous research in hardware reconfiguration in two main aspects. First, the syntax and semantics of the APIs of hybrid components in our system are preserved across the multiple instances, such that an application sees no difference (other than changes in quality and cost associated with a call) when versions are changed. This is accomplished through meta-programmed adapters, wrappers, and drivers that curtail variations in operation across the multiple versions with minimal overhead. Second, unlike other works that treat hardware/software reconfiguration as a monolithic operation, we split it into small steps executed while the operating system is idle. Hence, even with small available idle time, reconfiguration can be carried transparently and comply with real-time requirements.

III. Usage Model

Our approach assumes that concurrent applications are developed using threads. Each thread dynamically instantiates (and destroys) the components it needs, and the operating system automatically uses the component implementation that best suits its current needs. In X-Ware, multiple component implementations are available to applications through a single API. For the application, a call to a software function is, hence, identical to employing a hardware IP 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 implementations. 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 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 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.

In X-Ware, applications can be developed in C++ as any regular application. 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 parametrized class. 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.

IV. System Architecture and Implementation

The EPOS operating system was used as the backbone for the X-Ware API. EPOS is a multi-platform, component-based operating system that implements traditional operating system services through adaptable, platform-independent System Components [1]. EPOS supports aspects through a scenario adapter mechanism [1]. Distinct combinations of system components and scenario aspects lead to different software architectures. In this context, EPOS implements a framework that defines how components can be arranged together into a functioning system. EPOS framework, depicted in Figure 1, is realized by a C++ static metaprogram that is executed during component instantiation to adapt the component to coexist with other components as required by each application.

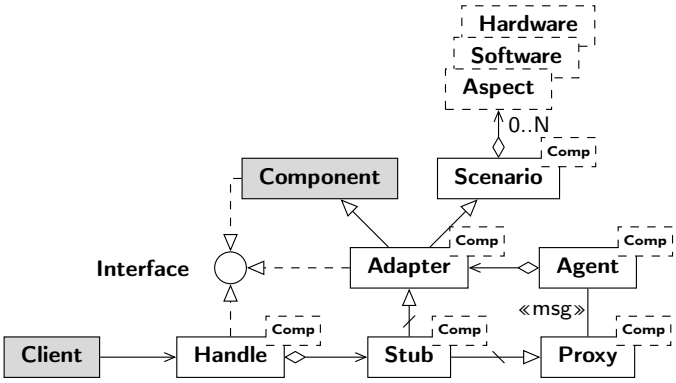


Fig. 1: EPOS component framework metaprogram.

`Stub` bridges `Handle` either with the abstraction's scenario `Adapter` or with its `Proxy`. The `Proxy` is used when the interaction between the component and the client crosses domains, such as SW/HW, user/kernel, or distinct machines on a network. Moreover, `Proxy` realizes the interface of the abstraction it represents, forwarding method invocations to its `Agent`. The `Agent`, likewise the `Handle` for a local scenario, forwards invocations to the abstraction's `Adapter`. The `Adapter` performs scenario adaptations for the corresponding abstraction, adapting its instances to perform in the selected scenario. It applies primitives supplied by the aspect program collection `Scenario` (e.g. remote or hardware) to abstractions, without making assumptions about the scenario aspects represented by these primitives. Adaptations are carried out by wrapping the operations defined by the component within the `enter()` and `leave()` scenario primitives, and also by enforcing a scenario-specific semantics for creating, sharing and destroying its instances.

The indirection between the client and the component in EPOS framework allows a transparent implementation of the X-Ware API. We modified EPOS so that `Handle` could be used as the switching point between implementations during runtime. `Handle` might point to any user-selected implementation while still maintaining internal references to other implementations for later usage. This way, during execution, only one component implementation is active. An operating system component called `Component_Manager` is charged with managing and monitoring system resources during runtime. The current implementation of a given component can be changed by `Component_Manager` by invoking `Handles's switch()` method. Figure 2 shows the resulting architecture generated by EPOS framework metaprogram for a component with two implementations. In this example, we have one software and one hardware implementation both aggregated by `Handle`. `Proxies`, `Agents` and `Adapters` were not depicted in the figure for simplicity.

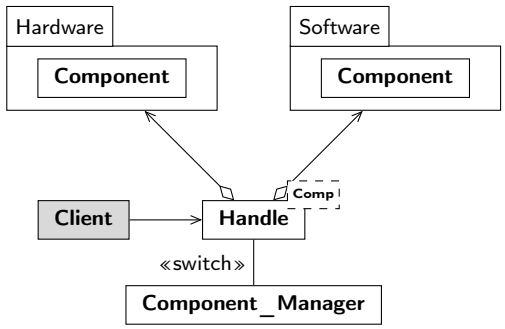


Fig. 2: Architecture generated by EPOS component framework metaprogram for a component with two implementations.

A. Implementation Selection

EPOS generic programming techniques enable the user to assign a priority to each component implementation in each application without inferring overhead, for instance, using EPOS traits mechanism. Traits are parameterized classes whose static constant members describe the properties (traits) of a certain type. A parameterized class `Traits<Component>` denotes the traits of the abstraction `Component` in a given application. It presents a metaprogrammed list of types populated by the user containing the preferred implementations of `Component`. The list order defines the implementation priority, p. Moreover, the application programmer might put only implementations that fulfill the application requirements in the list.

B. API Reconfiguration

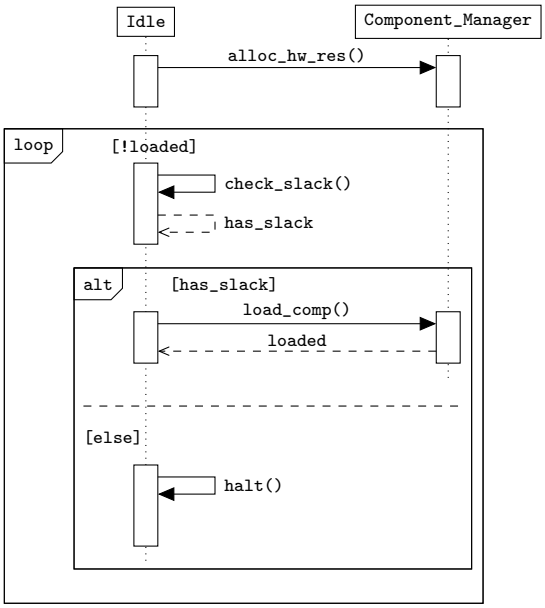
Implementation switch process is performed by the operating system in a speculative manner based on the defined policy. EPOS hard real-time scheduler [18] ensures that the *idle* thread is only scheduled when there is no other thread to run. The idle thread is, therefore, the entity in charge of keeping the system configuration tuned with the reconfiguration policy specified by the user. The speculative reconfiguration triggered by the idle thread is a key element in our approach but, ensuring that real-time threads will not have their deadlines compromised solely through it would require the reconfiguration procedure to be carried out without ever blocking the scheduler. Indeed, much of this procedure, which will be explained later on in details, can be performed in parallel with the execution of user threads, including reconfiguration policy enforcement and

bitstream loading. Nevertheless, the handling of a component's state and the redirecting of their clients to the new implementation are operations that have to be performed atomically.

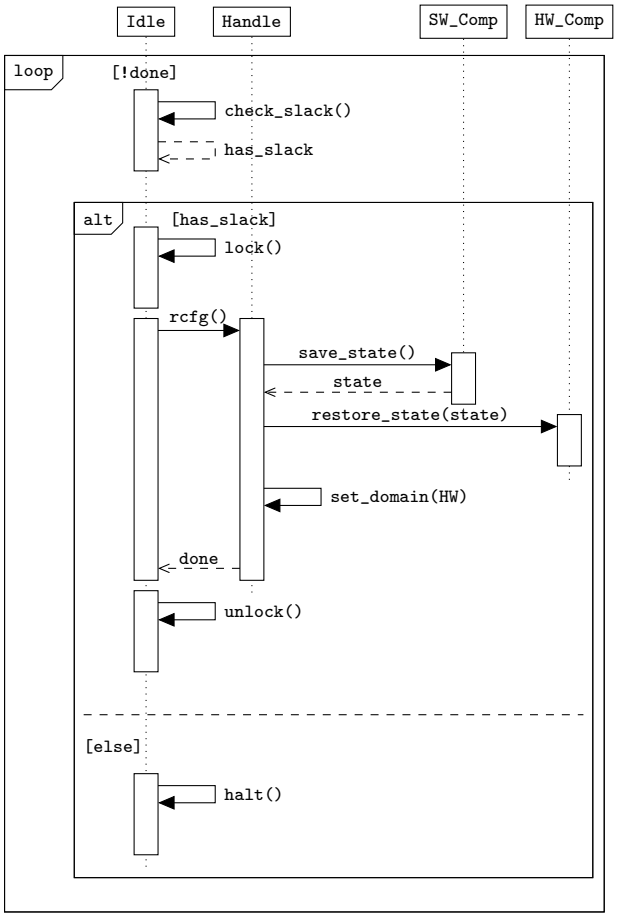
A second feature of EPOS scheduler useful for our API reconfiguration strategy is that it keeps the list of threads that are ready to be executed but did not yet reached their activation periods in an ordered, relative queue. Therefore, calculating the amount of time available for atomic reconfiguration activities becomes a deterministic operation that simply consists in subtracting the current time from the time stamp of the queue's head. Since both atomic activities, state and client handling, have deterministic duration (state handling is directly proportional to the component's internal state size, and client binding is a fixed cost operation), the idle thread can always determine whether a reconfiguration would compromise a deadline or not, postponing it in case it would. Combining these elements, we can ensure that a task set that was schedulable without our reconfiguration mechanism will still be schedulable when it is activated and will have a user-defined reconfiguration policy enforced speculatively whenever the execution of real-time threads yields some slack. We adopt the Least Laxity First (LLF) scheduling policy for thread scheduling because it agglutinates the slack time of *jobs* running for multiple periods, yielding intervals that are more likely to be suitable for reconfiguration operations. Nevertheless, other scheduling policies such as Earliest Deadline First (EDF) and Rate-Monotonic (RM) could be used.

To explain the reconfiguration process in details, let us assume a reconfiguration policy that favors hardware component implementations. That is, it will try to push as many components to hardware as possible. The process would be comprised of two main steps, which are illustrated by the sequence diagrams in Figure 3.

In the first step of the process, presented in Figure 3a, the idle thread loads a given component implementation into the reconfigurable hardware. Initially, the idle thread asks the `Component_Manager` if one of the system components may be reconfigured. If yes, the allocation of the hardware resources which will be used by the component is performed in `alloc_hw_res()`. For most reconfigurable platforms, it consists of querying the operating system for an available reconfigurable partition in the FPGA. Next, a bitstream containing the implementation of the given component to the available partition is loaded into the FPGA. Chunks of the bitstream are fetched and loaded to the bitstream each time `load_comp()` is invoked by the idle thread. The size of each chunk is adjusted to fit in the slack time available in the given scheduling period. `check_slack()` calculates if the available slack time is enough to transfer a minimal chunk of the component's bitstream. If the slack time is insufficient, the idle thread follow its original flow (usually halting the CPU). One could be tempted to assume that bitstream loading and user thread execution could be carried out in parallel, but since our focus is on hard real-time systems, such an assumption would rise complicated questions about arbitration in the system bus(es) that cannot be answered in an platform-independent way. We therefore check for amount of time remaining until the next thread activation. The first step ends



(a) Reconfiguration triggering and component loading.



(b) State handling and rebinding.

Fig. 3: Reconfiguration process sequence diagram.

when the bitstream in completely loaded.

In order to finalize the process, `Handle` must also update its domain in order to dispatch future method invocations to the correct component implementation. This step, depicted in Figure 3b, must be atomically executed and therefore requires enough slack time. When `check_slack()` indicates that the slack time is enough, initially the idle thread applies a lock to the scheduler. One might argue that if the slack time is enough, the idle thread will not be preempted during the whole step thus the lock is pointless. Nevertheless, on multicore machines, the idle thread must acquire the lock to prevent cross-core interference. Next, the idle thread invokes the `rcfg()` method of the chosen component's `Handle`. It first fetches the software component's state by invoking `save_state()` and further transfers it to the hardware component using `restore_state()`[1]. At this point the hardware component is ready to operate and its `Handle` change domain using `set_domain()`. From now on, the method invocations will all be dispatched to the component adapted to the hardware scenario.

The whole process is deterministic as we know the Worst Case-Execution Time (WCET) of each method. For most methods such as `check_slack()`, `alloc_hw_res()`, `set_domain()`, `lock()`, `unlock()` and `halt()` the process of obtaining its WCET is straightforward. Statistical methods based on static code analysis techniques and code profiling can be employed [19]. Nevertheless, for methods that rely on strict interaction with peripherals, the estimation might be trickier. For instance, `restore_state()` and `save_state()` execution time are component specific. It will vary according to the complexity of the component's internal state and bus access time but the WCET can also be estimated using statistical methods. As for `load_comp()`, it will be impacted by the underlying circuitry being used to reconfigure the fabric; Xilinx's commercial FPGAs rely on Internal Configuration Access Port (ICAP) or Processor Configuration Access Port (PCAP) while Altera's use Fast Passive Parallel (FPP). The WCET estimation for `load_comp()` will be based not only on the statistical methods mentioned previously but also on the reconfiguration bandwidth of the reconfiguration circuitry.

V. Results

We assembled a platform called EPOSSoC to support the deployment of the X-Ware API, Figure 4 shows its architecture. *Recfg nodes* represent the reconfigurable partitions that can contain a component in the hardware domain. As mentioned previously, the implementation of channels, proxies and agents can be realized in several different ways. We have chosen a Real-Time Star Network-on-Chip (RTSNoC) based interconnection scheme. A bus-based interconnect was discarded as it is not the most suitable choice for more heterogeneous designs in which hardware components have active roles [20]. The RTSNoC consists of routers with a star topology that can be arranged to form a 2-D mesh. Each router has eight bi-directional channels that can be connected to cores or channels of other routers. Each *Recfg node* is connected to a port of an *RTSNoC router* and one of the *RTSNoC routers* ports is connected to an AMBA bridge.

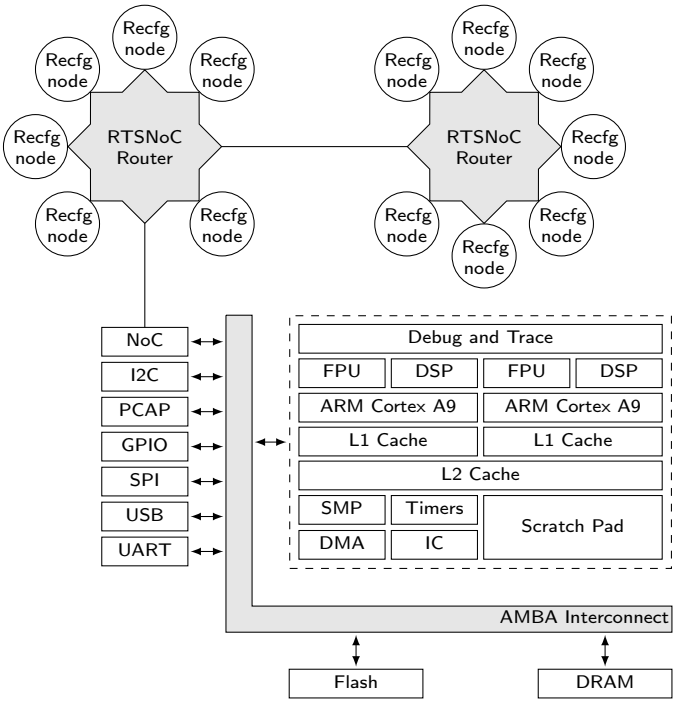


Fig. 4: Block diagram of the system.

Software implementations run on the dual-core Cortex-A9 processor executing EPOS (*CPU node*). The internal structure of the processor is based on the AMBA family of protocols. The *CPU node* provides the necessary run-time support to implement the proxies and agents described previously. Hardware reconfiguration is held by the Processor Configuration Access Port (PCAP) interface on the AMBA Interconnect, also managed by the operating system.

Platform-specific specializations of proxies and agents support the RTSNoC based communication scheme within EPOS. In software, `Component_Manager` keeps lists of all existing proxies to hardware and agents to software. Each component is associated with a unique identifier that is mapped by a resource table to a physical address in the NoC. Upon a call request, this address is used to build a packet containing the target method identifier and its arguments. An Interrupt Service Routine (ISR) defined in the `Component_Manager` handles the incoming packets. The ISR reads all pending packets and performs the necessary operations. When the manager receives a packet containing return values, it forwards the packet to the corresponding blocked proxy. When a packet contains data from a method call request, the information is forwarded to the dispatcher of the respective agent[2].

[1]Methods `save_state()` and `restore_state()` are implemented by the component's developers themselves (and not by the OS). Therefore they must agree upon a common representation of each component's state so both methods can interoperate.

[2]The exchange of pointers to data structures or data structures containing pointers is currently not supported mostly due to HLS tool limitations. The tool must be able to determine statically all structures to which a pointer might point. In our case, passing a pointer to a hardware component would mean sharing a data structure allocated to DRAM in the CPU node that is not known at synthesis-time.

We evaluated two components provided by the X-Ware API, the first is the Fast Fourier Transform Transform (FFT) algorithm and the second implementing the natural exponential function. Our goal was to reveal the possible trade-offs between implementations that might be explored by the `Component_Manager` to adapt the system's non-functional characteristics. A detailed decomposition of system overhead for reconfiguration can be found in a previous work [21]. The software components were compiled with GCC 4.7.2 targeting the ARMv7 ISA using optimization enabled by GCC's `-O` flag. The hardware platform was prototyped in ZedBoard, a development kit based on Xilinx's Zynq-7000, a SoC that couples a reconfigurable fabric with a hard core ARM Cortex-A9 dual core CPU. We used Xilinx's Vivado 13.4 for RTL hardware synthesis adjusting synthesis constraints to minimize circuit area.

A. Fast Fourier Transform

X-Ware API provides a Fast Fourier Transform Transform (FFT) to compute the Discrete Fourier transform (DFT) and its inverse. Behind the API, there are three implementations using the Cooley-Tukey algorithm. Two fo them are software implementation, `sw_flt_fft()` and `sw_dbl_fft()`, are Ne10 library [22] implementations based on a mixed Radix-2/Radix-4 algorithm. `sw_flt_fft()` and `sw_dbl_fft()` operate on float samples and double samples respectively. The hardware version, `hw_flt_fft()`, relies on Xilinx LogiCORE FFT IP [23] that pipelines several Radix-2 butterfly processing engines to offer continuous data processing. A DMA engine moves data in and out of the FFT IP through Zynq's Accelerator Coherency Port (ACP). ACP provides to hardware accelerators access to the DDR memory with L2 cache coherency.

We measured the direct transform execution time of all implementations. The FFTs were performed on 1024 double-precision floating point samples, which were type-casted to single-precision floating point before being used in `sw_flt_fft()` and `hw_flt_fft()`. Implementing floating-point operations on an FPGA can be expensive in terms of the resources required. Xilinx FFT IP converts incoming floating point samples to fixed-point, utilizes a fixed-point FFT internally and converts the results back to floating point to achieve similar noise performance to a full floating-point FFT, with significantly fewer resources. The operations were repeated 1000 times to measure the average execution time. The CPU clock frequency is 666.666 MHz while FFT IP clock frequency is 150 MHz. The time measurements were made using the Zynq's 64-bit Global Timer. The results are presented in the first column of Table I. `hw_flt_fft()` is almost 18 times faster than `sw_dbl_fft()` when processing 1024 samples. The performance boost can be explained by the parallel implementation of the FFT butterflies coupled with the DMA engine to move to and from the IP. Moreover, the L2 cache coherency speeds up DMA fetching of samples lowering DMA memory access time.

Our following analysis was the power consumption of Zynq SoC while executing each FFT implementation. It is not possible to measure Zynq's energy consumption in ZedBoard, only energy consumed by the whole board. To overcome this issue we estimated the energy consumed by the board peripherals by putting Zynq in low power mode and measuring the board power consumption. The gathered value, 3.132 W, was subtracted from all other measurements to obtain only Zynq power consumption. The second column of Table I presents the measured consumed energy values for ZedBoard executing the three FFT implementations in a loop. The energy was calculated by multiplying the measured power with the execution time. `hw_flt_fft()` consumes almost 20% more power than the `sw_flt_fft()` due to the dynamic power consumption in the FPGA. While software implementations do not utilize the reconfigurable resources, the FPGA consumes only static power. As `hw_flt_fft()` depends on a hardware IP, its dynamic consumption accounted in the power measurement. Nevertheless, `hw_flt_fft()` is much faster than `sw_dbl_fft()` and `sw_flt_fft()` resulting in a smaller energy consumption.

TABLE I: FFT implementations characteristics.

Implementation	Time (μs)	Energy (μJ)	RMSD	MAPE
`sw_dbl_fft()`	701.817	833.759	0.0	0.0
`sw_flt_fft()`	592.728	697.048	1.5641e-04	9.4122e-05
`hw_flt_fft()`	40.389	57.675	1.5649e-04	9.4232e-05

Moreover, we compared the accuracy of `sw_flt_fft()` and `hw_flt_fft()` with the higher precision FFT implementation, `sw_dbl_fft()`. To estimate accuracy we calculated the root-mean-square deviation and the mean absolute percentage error of the results datasets taking `sw_dbl_fft()` results as the reference data set. The third and fourth columns of Table I present the calculated results. `hw_flt_fft()` is slightly less accurate than `sw_flt_fft()` due to its fixed-point processing phase nevertheless, both are considerably less accurate than `sw_dbl_fft()`.

`hw_flt_fft()` is arguably the implementation that better performs both in execution time and energy consumption but, it depends on hardware resources that might not always be available (other components might be occupying the reconfigurable fabric). In such cases, `sw_flt_fft()` might be used if its lower precision compared to `sw_dbl_fft()` is tolerable by the application due its to its smaller energy consumption.

B. Natural Exponential

We also profiled different implementations of five natural exponential function implementations; the results are presented in Table II. `hw_exp()` utilizes Xilinx's single precision Floating-Point Operator IP [24] operating at 100 MHz on the FPGA. The function argument is written to a memory mapped register by the CPU, processed by the IP and read back. `exp()` and `expf()` are GNU's libm[25] implementation, they operate on double and single precision floating point arguments respectively. `fastexp()` and `fasterexp()` are approximate implementations present in fastapprox [26] library, both operate on single precision arguments.

Each implementation processed 1000 arguments ranging from -32.0 to 32.0. The results presented in the first column of Table II show that the faster approximated implementation is three times faster than `exp()`, the only implementation

TABLE II: Natural exponential implementations characteristics.

Implementation	Time (μs)	Energy (μJ)	RMSD	MAPE
`exp()`	0.357	0.450	0.0	0.0
`expf()`	0.345	0.435	4.13e06	3.116e-07
`fastexp()`	0.191	0.241	2.28e08	2.280e-05
`fasterexp()`	0.117	0.152	1.50e11	1.530e-02
`hw_exp()`	1.280	1.567	4.13e06	3.108e-07

operating on double precision arguments. The communication overhead between software and hardware overwhelms the execution time of exponential operation itself. Such small granularity operations are faster in software when compared to its hardware counterparts running at smaller clock frequencies.

We evaluated the accuracy of all natural exponentiation implementations by comparing them with the most accurate implementation, `exp()`, operating with double precision floating point inputs and outputs. Table II contains the root-mean-square deviation and the mean absolute percentage error of each implementation results. As they are based on approximated calculations, `fastexp()` and `fasterexp()` present a bigger RMSD and MAPE compared to other version, being thus less accurate. `hw_exp()` does not fully comply with IEEE Standard for Floating-Point Arithmetic as the deviations provide a better trade-off between resources against functionality [24]. That is why its MAPE is not equal to `expf()` MAPE.

The energy consumption of the chosen natural exponential implementation is inversely proportional to its accuracy except for `hw_exp()`; it underperforms all other implementations in energy consumption and execution time. However, it still might be a valuable implementation option when the CPU is under heavy load or with a higher priority task scheduled, and the operating system wishes to transfer part of this load to the FPGA.

VI. Conclusion

In this paper we introduced X-Ware, a framework for *mutant* computing substrates. X-Ware delivers to applications APIs that may be realized through many different implementations, ranging from high-quality software versions to software approximations, cloud offloaders, and hardware accelerators. While the syntax and semantics of the API 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. We demonstrated X-Ware with library of mathematical functions, and showed how the framework can help applications trade-off quality for energy efficiency (up to 93% savings) and performance (94% decrease in execution time).

While in our demonstrations for this paper all components were stateless, unlike previous work [6], X-Ware is not restricted to stateless components and pure functions. In future work we intend to automate some of the state migrations between component implementations, but in our current framework state migrations must be handled by the component designer. In future work we also intend to explore automatic and compiler-aided component generation. Finally, we will expand our current work on the component mutation governor to account for different system optimization goals at runtime such as energy savings, mitigation of silicon aging, and real-time performance.

References

[1] A. A. Fröhlich, "Application-oriented operating systems," Ph.D. dissertation, Technical University, Berlin, 2001, Ph.D. Thesis.

[2] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "PetaBricks: A language and compiler for algorithmic choice," *SIGPLAN Not.*, vol. 44, pp. 38–49, 2009.

[3] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger, "Eon: a language and runtime system for perpetual systems," in *SenSys*, 2007.

[4] W. Baek and T. M. Chilimbi, "Green: a framework for supporting energy-conscious programming using controlled approximation," *SIGPLAN Not.*, vol. 45, pp. 198–209, June 2010.

[5] A. Lachenmann, P. J. Marrón, D. Minder, and K. Rothermel, "Meeting lifetime goals with energy levels," in *SenSys*, 2007.

[6] L. Wanner and M. Srivastava, "ViRUS: Virtual function replacement under stress," in *Proc. of the 6th USENIX Conference on Power-Aware Computing and Systems*, ser. HotPower'14. USENIX, 2014.

[7] M. J. Voss and R. Eigemann, "High-level adaptive program optimization with adapt," *SIGPLAN Not.*, vol. 36, no. 7, pp. 93–102, Jun. 2001. [Online]. Available: http://doi.acm.org/10.1145/568014.379583

[8] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," *SIGARCH Comput. Archit. News*, vol. 39, pp. 199–212, 2011.

[9] H. Falaki, "Automating personalized battery management on smartphones," Ph.D. dissertation, UCLA, 2012.

[10] A. Pant, P. Gupta, and M. v.-d. Schaar, "AppAdapt: Opportunistic application adaptation to compensate hardware variation," *IEEE Transactions on Very Large Scale Integration Systems*, 2011.

[11] J. M. Arnold and D. A. Buell, "VHDL programming on Splash 2," in *Intl. Workshop on Field Programmable Logic and Applications on More FPGAs*, 1994, pp. 182–191.

[12] C. Bobda, M. Majer, A. Ahmadinia, T. Haller, A. Linarth, and J. Teich, "The Erlangen Slot Machine: Increasing flexibility in FPGA-based reconfigurable platforms," in *International Conference on Field-Programmable Technology*, 2005.

[13] D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp, "Achieving programming model abstractions for reconfigurable computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 34–44, January 2008.

[14] R. Brodersen, A. Tkachenko, and H. K.-H. So, "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH," in *CODES+ISSS*, October 2006, pp. 259–264.

[15] A. Ismail and L. Shannon, "FUSE: Front-end user framework for O/S abstraction of hardware accelerators," in *Intl. Symposium on Field-Programmable Custom Computing Machines (FCCM),*, 2011.

[16] E. Lubbers and M. Platzner, "ReconOS: Multithreaded programming for reconfigurable computers," *ACM Transactions on Embedded Computing Systems*, vol. 9, no. 1, pp. 8:1–8:33, October 2009.

[17] Y. Wang, X. Zhou, L. Wang, J. Yan, W. Luk, C. Peng, and J. Tong, "SPREAD: A streaming-based partially reconfigurable architecture and programming model," *TVLSI*, vol. 21, no. 12, pp. 2179–2192, 2013.

[18] G. Gracioli, A. A. Fröhlich, R. Pellizzoni, and S. Fischmeister, "Implementation and evaluation of global and partitioned scheduling in a real-time OS," *Real-Time Systems*, vol. 49, no. 6, pp. 669–714, 2013.

[19] R. Wilhelm and et. al., "The worst-case execution-time problem — overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 36:1–36:53, May 2008.

[20] G. De Micheli, C. Seiculescu, S. Murali, L. Benini, F. Angiolini, and A. Pullini, "Networks on chips: From research to products," in *Proc. of the 47th Design Automation Conference*, 2010, pp. 300–305.

[21] J. G. Reis, A. A. Fröhlich, and L. Wanner, "A framework for dynamic real-time reconfiguration," in *18th EUROMICRO Conference on Digital System Design*, Funchal, Portugal, Aug. 2015.

[22] Project Ne10 developers, "Project Ne10," [Online] Available: https://github.com/projectNe10/Ne10, 2015.

[23] *LogiCORE IP Fast Fourier Transform v8.0*, Xilinx, 7 2012.

[24] *LogiCORE IP Floating-Point Operator v6.0*, Xilinx, 1 2012.

[25] Free Software Foundation, "GNU C library," [Online] Available: http://www.gnu.org/software/libc/, 2014.

[26] P. Mineiro, "fastapprox software library," [Online] Available: https://code.google.com/p/fastapprox/, 2014.