Aaron: An Adaptable Execution Environment

Marc Brünink, André Schmitt, Thomas Knauth, Martin Süßkraut, Ute Schiffel, Stephan Creutz, Christof Fetzer
Technische Universität Dresden
Department of Computer Science; 01062 Dresden; Germany
{marc, andre, thomas, suesskraut, ute, stephan, christof}@se.inf.tu-dresden.de

Abstract—Software bugs and hardware errors are the largest contributors to downtime [1], and can be permanent (e.g., deterministic memory violations, broken memory modules) or transient (e.g., race conditions, bitflips). Although a large variety of dependability mechanisms exist, only few are used in practice. The existing techniques do not prevail for several reasons: (1) the introduced performance overhead is often not negligible, (2) the gained coverage is not sufficient, and (3) users cannot control and adapt the mechanism.

Aaron tackles these challenges by detecting hardware and software errors using automatically diversified software components. It uses these software variants only if CPU spare cycles are present in the system. In this way, Aaron increases fault coverage without incurring a perceivable performance penalty. Our evaluation shows that Aaron provides the same throughput as an execution of the original application while checking a large percentage of requests — whenever load permits.

Keywords—Fault detection; Fault tolerance; Diversity methods; Adaptive algorithm; Compiler transformation

I. INTRODUCTION

More and more, our daily life depends upon computing systems. The proliferation of those systems is accompanied by a demand for security, safety, and availability. To satisfy these demands a large variety of dependability mechanisms have been developed, both using either hardware or software solutions.

Hardware solutions to dependability issues are costly to develop and deploy; they are a good choice for techniques that are mature. One example is the NX bit used by W’X page protection (every memory page is either writable or executable per default). For dependability mechanisms that might be modified, most hardware solutions lack adaptivity. Techniques that are useful for only a minority of users are unlikely to be integrated into COTS hardware. Building specialized hardware incorporating these techniques can result in a prohibitively high cost-performance ratio compared to COTS components. Thus, dependability mechanism should be implemented in software until they have matured and have been proven to be useful in a majority of application scenarios.

In contrast to COTS hardware, software solutions are highly adaptable. Furthermore, many different software solutions that cope with dependability issues are available. Those include, but are not limited to: out-of-bounds checker [2], redundant execution [3, 4], software encoded processing [5], and recovery blocks [6]. Each of these different approaches target a specific set of failures; however, none covers all failures observable in deployed systems. Although coverage can be increased by multiplexing dependability mechanisms, the overheads of different mechanisms add up. Even worse, interactions between them can lead to overheads larger than linear. In addition, multiplexing might have a negative effect on the coverage of the individual mechanisms.

Some of the existing dependability mechanisms only solve problems in specific execution environments or programming languages. Naturally, the question arises, why one should apply a dependability mechanism instead of changing the underlying setup. For example, instead of deploying an out-of-bounds checker, a software engineer could simply use a programming language that is safe in respect to out-of-bounds accesses. Similar arguments can be applied to dangling references (garbage collected languages) and bit flips (redundant hardware). However, changing the programming language or execution environment in these ways only shifts dependability mechanisms to lower levels. Although shifting might enable stronger optimization or more efficient implementation, it also decreases adaptability. Adaptability is favorable because it can lead to higher efficiency in the long run: As more mature software is deployed, less errors are present in the software; but, the cost of checking is constant. As a result the cost-benefit ratio gets worse. Using a traditional deployment it is not possible to downscale checking. For example, once an application is developed in a safe language, it is hard to loose the incurred constraints during runtime in order to increase performance.

Aaron tackles these challenges by scheduling different runtime checks dynamically depending on the load of the system and the maturity of the software. Maturity is not a monotonically increasing property but fluctuates especially at major releases. To this extent Aaron has to adapt, and, potentially, take hints from a system administrator.

The different software diversity mechanisms we used to increase safety and security are discussed in Section II-B. Aaron uses CPU spare cycles to schedule software variants...
dynamically, which is detailed in Section II-D. In Section III we evaluate the system and show that it adapts to changes in system load instantaneously.

The resulting system can: (1) use dependability mechanisms that exhibit a high overhead; (2) adapt failure coverage at runtime; (3) incorporate hints given by a system administrator.

II. THE AARON APPROACH

In many environments, availability and costs are ultimately more important than security and safety. With the exception of highly critical systems, we often take the potential risk of a security or safety violation with the goal to increase availability or to decrease costs. For example, triple modular redundancy or byzantine fault tolerant systems are often not used because the costs outweigh the benefits. Aaron targets these environments in which safety and security are not the prime objectives but are desired nevertheless. In the following, we enumerate the design goals of Aaron and explain how they are fulfilled:

1) **Goal**: Hardware and software errors are detected without additional hardware and without increasing costs of software development.
   **Approach**: Aaron uses software diversity to detect and, if possible, tolerate errors in deployed systems. It generates software variants automatically without user intervention. Aaron does not need special hardware. It runs on COTS hardware making it especially useful in scenarios in which high fault coverage is desired but without additional cost due to special purpose hardware.

2) **Goal**: The cost of processing is increased at most moderately.
   **Approach**: Aaron exploits spare cycles present in the system. No additional hardware has to be deployed to cope with increased load caused by error detection. Furthermore, using spare cycles for error detection increases power costs only moderately since common computers consume a significant amount of power even while being idle.

3) **Goal**: Error detection does not influence system throughput.
   **Approach**: The runtime overhead of Aaron itself is negligible. Aaron adapts extremely fast to the current load situation. Error detection is performed on a best-effort basis; if in doubt, Aaron opts for throughput instead of error detection.

To build a solid foundation for further discussion, we continue with a general overview of the architecture of Aaron. Subsequently, we examine specific details of Aaron to gain insights into its inner workings. We use the obtained knowledge to discuss Aaron’s limitations at the end of the section.

Figure 1. Aaron augments the existing applications by scheduling automatically diversified software.

A. Architectural Overview

Aaron augments an application with software dependability mechanisms. The software developer embeds Aaron into the application using a very slim interface. To this end, three parts of the application have to be isolated: input streams, tasks, and worker threads.

The developer has to re-route the input streams of tasks to Aaron; in our experience a very straightforward endeavor. A task is a small unit of work the application should perform. It subsumes multiple different concepts including, but not limited to, jobs, events, kernels, and requests. Especially applications that distribute computation across multiple nodes show the necessary structure. After reading the task from an input channel (cf. Figure 1, step 1), the application passes it to Aaron using a simple function call. The function call is added by the software developer. Optionally, the application might perform minor pre-processing before a task is passed to Aaron, e.g. tasks might be prioritized.

Next, Aaron inspects current system conditions. Depending upon system load, Aaron chooses a software variant from a variant pool (step 2). A variant is a version of the application that was diversified at compile time and protects the execution of the task against a specific set of errors. We present more details in the next section. Aaron forwards the task and a function pointer reflecting the chosen variant to a worker thread (step 3). The thread processes the request using the determined variant. Output is sent via an arbitrary application-specific channel, which is not depicted in the Figure.

B. Automatic Diversity

Diversification is a common practice to detect and mask errors in deployed systems, for example in avionics. However, manual diversification, e.g. design diversity, is costly in general and only justifiable in highly critical areas. Aaron uses automatic software diversification to achieve high fault coverage without increasing development costs.
A large number of software diversification methods are available (e.g. [7, 4, 8, 9, 10, 11, 12]). In principle, Aaron can use any of those. We implemented multiple software diversification methods (Table I). The goal is neither to present new diversification methods nor to extend knowledge about existing methods. Instead, we use them to prove the applicability and the overall soundness of Aaron. Aaron can be easily extended with new diversification methods if desired.

The diversification itself is achieved using the LLVM compiler infrastructure [13]. LLVM translates the source code of the application into an intermediate representation (IR). For each diversification method, the intermediate representation is copied and augmented with runtime checks (Figure 2). All functions are renamed and calls and references are adjusted accordingly. Global variables are shared among variants. Finally, all variants and the global variables are linked into a single file. Thus, we gain multiple versions, i.e. variants, of the application. Each variant protects against a specific set of errors. All variants together form the diversified application.

C. Power Consumption

Computers consume a significant amount of power even while being idle or only lightly loaded. Figure 3 shows the power consumption of one of our Dell Precision R5400 machines, measured with the help of a Raritan Dominion PX-5528 power distribution unit.

If the node is not loaded at all, it already consumes 133 watts. Power consumption increases linearly and peaks at 190W. At a load of 50%, power consumption is already 88% of consumption under full load (167 watts). Only a small increase of about 14% has to be paid in order to use the remaining 50% of CPU power.

To handle sudden load surges, cluster deployments are often oversized [16]. Aaron uses those spare cycles present in deployed systems to schedule different diversified software variants to detect and tolerate runtime failures. Since Aaron uses spare cycles, we expect only a small increase in total power consumption.

D. Scheduling Software Variants

To be able to exploit spare cycles in deployed systems without generating any user-perceivable runtime penalty, Aaron has to adapt to varying workloads extremely fast. Aaron currently relies on an important fact of applications running in cluster environments: cluster applications process

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>An unchanged version of the application. However assertions do not trigger an abort. Instead, violated assertions are logged and ignored.</td>
<td></td>
</tr>
<tr>
<td>Assertion</td>
<td>Violated assertions cause an abort of the application.</td>
<td></td>
</tr>
<tr>
<td>OverAllocate</td>
<td>Pads stack and heap allocations to tolerate present buffer overflows. Stack allocations are padded on a per object basis, so each stack variable is padded.</td>
<td>[8, 9]</td>
</tr>
<tr>
<td>NullWrite</td>
<td>Ignores writes to address 0x0.</td>
<td>[14]</td>
</tr>
<tr>
<td>NullRead</td>
<td>Return 0 if address 0x0 is loaded.</td>
<td>[14]</td>
</tr>
<tr>
<td>NullDeref</td>
<td>Combines NullWrite and NullRead.</td>
<td></td>
</tr>
<tr>
<td>SWIFT</td>
<td>SWIFT duplicates all instructions and registers apart from memory accesses and control flow instructions to detect transient hardware errors. Since the original implementation of SWIFT was not available, we reimplemented SWIFT.</td>
<td>[15]</td>
</tr>
<tr>
<td>SWIFTCFC</td>
<td>SWIFT alone does not detect control flow errors. SWIFTCFC adds control flow checking to SWIFT.</td>
<td>[4]</td>
</tr>
</tbody>
</table>
workload in parallel. As a result the workload is divided into
different tasks by the application developer. For example,
in web and database servers, batch and event processing
systems one can discover the concept of a task. Aaron
exploits this task orientation: it decides whether to apply
checking on a task by task basis. Aaron targets task-oriented
applications, which are prevailing in cluster environments.

1) Load Estimation: The scheduler’s job is to decide
which variant to use. This decision is based on the current
load of the system. Since CPU utilization is a local attribute,
calculation of system load is performed locally on each
cluster node.

The load calculation has to be precise and accurate. We
implemented several calculation methods.

The first approach estimated system load by inspecting
the actual CPU load of the node. Although it is straight-
forward and seems to be the natural approach, CPU load is
not precise enough: In our experience the scattering of values
renders fine-grained, accurate variant selection impossible.

In the second approach we simply inspect the task queue
and use the number of pending tasks to estimate system
load. This mode proved to be highly efficient and accurate.
It is best suited for applications in which clients issue tasks
in an asynchronous manner. Asynchrony ensures that tasks
are not queued at the client but at the server, and the work
queue of the server reflects the actual system load.

As a third option we calculate system load using the ratio
of pending IO sockets per active client. This is best suited
for applications that use synchronous tasks.

An application developer can use one of the load estima-
tors provided by Aaron or build a custom estimator.

2) Best-effort Detection: In contrast to existing methods,
Aaron uses a best-effort approach to error detection: Aaron
does not guarantee a specific error coverage. Instead, it
observes error behavior and maximizes coverage using spare
cycles present in the system.

Before a task is processed by the system, Aaron gathers
the current system load \( \text{load}_c \) and compares it with a user-
specified target load \( \text{load}_t \). If the current load is higher than
the target load, Aaron plays safe and uses the unmodified,
original software. Otherwise it schedules software variants
that detect errors. Aaron chooses the software variant \( v \) from
the set of all variants \( V \) with an overhead \( o(v) \) closest to a
target overhead \( t \):

\[
v = \begin{cases}
  v_{\text{noChecks}} & \text{if } \text{load}_c \geq \text{load}_t \\
  v_i : d(v_i) = \min\{d(v_j) | v_j \in V \} & \text{else}
\end{cases}
\]

with

\[
d(v) = \left| o(v) - t \left( \frac{\text{load}_c}{\text{load}_t} \right) \right| .
\]

Aaron scales the utilization factor \( \frac{\text{load}_c}{\text{load}_t} \) to the interval of
all observed overheads. The mapped value is used as the
target overhead (Figure 4). The variant that is closest to
the target overhead is used to process the current task. In
our experience, there is typically a correlation between the
percentage of errors detected and the overhead of a vari-
ant. Therefore, Aaron chooses the variant with the highest
overhead without overloading the system.

The simplicity of the scheduling is a main feature of
Aaron and enables it to fulfill the design goal of having low
overhead (Section III). However, to determine the overhead
of variants at runtime we introduce randomness into the
scheduling process.

3) Dynamic Overhead Calculation: The scheduler uses
the overheads of the different variants to choose the variant
most appropriate at the current system load.

To facilitate accurate variant selection, the estimated over-
heads should reflect the actual overhead with low variation.
The overhead a variant exposes during processing of a spe-
cific task is influenced by the executed code path. Since the
code paths may differ, the mixture of executed instructions
might differ, too. For example, if a large percentage of
all executed instructions are memory reads, the NullRead
variant has to protect a relatively large fraction of the
execution. Thus, NullRead will expose a relatively high
overhead. However, if memory reads are comparatively rare
the overhead is lower. In sum, overheads are not static values
but depend on the executed task.

For a most accurate estimation of overheads, one has to
map each task to an overhead. This involves mapping multi-
ple dimensions of a task (e.g. operation requested, task size)
onto a single value. Unfortunately, this accurate prediction
consumes quite some processing power. Aaron eludes this
conflict by assuming an unimodal distribution of overheads
with low variance. Aaron handles changing workloads and
thus changing overheads by measuring them at runtime. It
exploits the assumption of an unimodal distribution by using
the median of observed values.

For each variant, Aaron keeps a sliding window con-
taining the execution times of the last tasks processed
using this variant. The sliding window is kept up-to-date by measuring the execution time of each processed task and replacing values in FIFO order. To make sure values are up-to-date even for variants that are not used at all, Aaron samples variants regularly. To this end, each variant has a logical timeout. The internal clock is driven by the processing of tasks; whenever a task is processed, the clock is incremented. For each processed task, Aaron measures the execution time. If the measured time and the estimated execution time for the used variant are reasonably close, the timeout is reset. Otherwise, if both values diverge more than a threshold, the timeout is decreased by the difference between the current clock value and the last time the timeout was reset. This fast-tracks the re-sampling of the current variant if needed. The mechanism ensures that (1) variants are regularly executed to verify and update the estimated overhead; thus, minor changes in the perceived overhead are gradually reflected in the estimations. (2) If verification fails the variant is re-sampled in a timely manner; a sequence of major mispredictions leads to a rapid adaptation of new values.

E. Fault Coverage

The probability that a fault is detected alias total fault coverage $C_t$ depends upon the coverage of the single variants, $C_v$, and the probability $P_v$ that a single variant is selected for execution:

$$C_t = \sum_{v=0}^{\left| V \right|} C_v \times P_v.$$  

Since $\sum_{v=0}^{\left| V \right|} P_v = 1$ and $0 \leq C_v \leq 1$:

$$C_t \leq \max\{\{C_v|v \in V\}\}.$$  

The actual fault coverage of Aaron, $C_t$, is less than or equal to the fault coverage of the best software variant.

In theory, total coverage can be maximized by using the software variant with the highest coverage. Unfortunately, it is hard at best to estimate the coverage of different variants. Fault coverage not only depends upon the system and the software variants, it also changes over time, for example due to an altered workload. Furthermore, using the variant with the highest coverage might impose a high overhead all the time.

Aaron maximizes fault coverage by adapting the error detection to the current workload. Aaron always tries as hard as possible to detect errors — but not harder.

F. Limitations of Aaron

Aaron is an infrastructure for early deployment and constant monitoring of cluster applications. It monitors execution using software diversity to find errors pro-actively. In the following, we summarize the constraints, which should be fulfilled by applications using Aaron. They result directly from the previously presented design decisions:

1) The computation of the application is split into tasks operating on the state of the application.
2) The level of the current workload can be estimated.
3) The workload of the application is throughput oriented. Latency is not as critical as throughput.
4) Processing of tasks should be isolated from each other. Complex interdependencies between tasks make it harder to use Aaron since these interdependencies have to be understood and taken care of manually.

These limitations are typically fulfilled by applications targeting cluster environments. Aaron also targets cluster environments.

Most cluster applications are task-oriented and tasks can be processed in parallel. Since interdependencies limit parallelism, those applications are tuned to eliminate them as far as possible. Although there might be a hard bound on latency, throughput is often more important. Aaron uses multiple queues to organize the pending tasks. As a direct consequence Aaron might reorder tasks unintentionally, and latency might fluctuate. Aaron could use priority queues to minimize the latency. However, this is an orthogonal research question and out of scope of current work.

Aaron adds calculations to the processing in order to find errors. These additional calculations are not free but cost processing time. As a result, Aaron increases the latency even if the queues are empty and new requests are processed immediately. Automatic parallelization of runtime checks [17] can be used to minimize the increase in latency.

III. Evaluation

We performed several experiments, each designed to answer one of the following questions:

1) **Power consumption**: How much additional power is consumed by Aaron’s fault detection?
2) **Scheduling overhead**: How large is the overhead of Aaron? Is the design goal of having negligible overhead fulfilled?
3) **Spare cycles**: How good does Aaron exploit spare cycles in the system?
4) **Throughput**: How many requests can be processed by Aaron? Does Aaron influence maximum throughput negatively?
5) **Responsiveness**: How does Aaron react to sudden load changes? How fast can the system adapt the fault detection to load changes? How does the workload influence the selection of variants?

A. Setup

We use several different applications to answer the questions posed. The applications are chosen in such a way that the behavior of Aaron in different situations becomes apparent.
Wordcount
The client sends a chunk of text to the server. The text is drawn randomly from a 178MB large document. The server counts the occurrences of each word in this text. We experimented with different sizes for the chunks. The size did not influence the behavior of Aaron; we fixed it arbitrarily at 20KB. Wordcount is bound by network throughput.

MD5
The request contains between 100 to 100k random bytes. For each request, the application computes 15 MD5 hash sums. We designed this artificial application to be able to evaluate Aaron’s behavior for a very CPU intensive application: MD5 is CPU bound; throughput will be hindered by available computational power on the node long before the network is saturated.

Zoologist
This application is inspired by the distributed coordination service developed at Yahoo called ZooKeeper [18]. Zoologist provides operations for manipulating a distributed name space. Requests can add and delete nodes in a hierarchical name space. Each name space node can also contain data. The state is kept in a hash map. All operations require little more than a simple lookup in the hash table. This makes the throughput for this application network bound. We implemented Zoologist because ZooKeeper uses Java. Currently Aaron only supports C/C++.

LibPNG
This service receives as a request the name of a file. To be able to evaluate Aaron’s behavior for applications accessing local disk, the file is stored locally on the server running the application. The application calls a test suite function from the LibPNG library. The test suite performs a set of image manipulations. Because image file formats are complex, image handling libraries are especially prone to programming errors. This makes LibPNG a suitable candidate to evaluate its error handling capabilities.

The experimental setup is as follows: our cluster consists of multiple nodes connected via Gigabit Ethernet. Each machine has two Intel Xeon E5405 CPUs with four cores each and 8GB physical RAM. There is one hard disk per node attached via SATA2 and spinning with 7200RPM. Each node runs Debian Linux 5.0 with kernel 2.6.26.

Unless stated otherwise, all measurements represent the truncated arithmetic mean of five runs.

B. Power Consumption
One of the motivations for this work is, that cluster nodes consume a significant amount of power even while being idle or only lightly loaded (Section II-C). Aaron exploits these spare cycles for fault detection.

We measured the increase in power consumption of the MD5 application for different workloads (Figure 5). Without error detection (denoted as Native in the Figure), processing 400 tasks per second consumes 145W (54% CPU utilization). Using Aaron for error detection increases power consumption to 185W (97% CPU utilization). An increase of 28%.

C. Scheduling Overhead
To get a baseline measurement of system performance, we implemented a Null-application. This application reads empty tasks from the IO sockets and returns an empty answer to the client. It does not process tasks in any way. It is a good way to measure how many requests our system can process in theory, and how much processing overhead the framework itself adds.

Figure 6 shows the CPU utilization for processing null-requests. It compares the execution of Aaron with runtime checking with a run without runtime checking (labeled Native in the Figure). While processing 100k requests, roughly two cores are busy reading requests from the TCP socket, putting them in the appropriate processing queue, and running the scheduling algorithm. Since the null-application does not process any tasks, the minor difference between the two curves reflects the overhead of Aaron’s scheduling algorithm. Because the curves are virtually identical, we can deduce that the scheduler is indeed sufficiently fast and fulfills the design goal of negligible overhead.

D. Using Spare Cycles for Fault Detection
To evaluate how good Aaron exploits spare cycles, we measured the CPU utilization for varying workloads and different applications (Figure 7).

Aaron uses spare cycles to schedule runtime checks: In Figure 7, the CPU utilization is considerably increased for all applications. Using Aaron the utilization of LibPNG and
Zoologist decreases after 4k and 20k requests per second, respectively. As the request rate increases, Aaron plays it safe: It schedules software variants with lower overhead.

MD5 is CPU bound. Even with a low request rate, CPU utilization is already above 50%. As soon as workload reaches 1800 request per second, no more spare cycles are present in the system.

For applications that are not CPU bound, e.g., Zoologist and LibPNG, CPU utilization of the native version does not reach the maximum. Zoologist is network bound, as the computational overhead per operation is small. LibPNG has a high contention rate and reaches peak throughput with low CPU utilization (about 25%) at 8000 requests per second.

E. Throughput

Throughput is usually a most important factor of a cluster application. Figure 8 shows the throughput for a varying rate of incoming requests.

MD5 and LibPNG reach the point of saturation at about 1800 and 8000 requests per second, respectively. MD5 is very CPU intensive; whereas LibPNG puts some load on the file system for accessing PNG images. Wordcount saturates at about 5700 requests per second. Each request processes 20KB, resulting in an accumulated throughput of about 111MB per second. Using the tool iperf, we measured the maximal throughput of TCP as 115 MBps, verifying that Wordcount is indeed network bound.

The most important observation of this measurement is that Aaron barely has any effect on the throughput; for all applications it is indistinguishable with a small deviation for LibPNG: the throughput is slightly higher at 12k requests per second when using Aaron. We attribute this to random fluctuation in the measurements.

F. Responsiveness

So far we have shown that Aaron’s overhead is negligible (Section III-C), and throughput is not influenced by using spare cycles (Section III-E). Next, we will show that Aaron adapts extremely fast to changes in the workload.

We evaluated Aaron’s speed of adaptation using the following setup: We used the CPU-bound MD5 application
and put on a workload of 1000 tasks per second. This utilizes the system to 60% of its peak throughput. At time $t_1$, the workload generators increase the request rate to 10k tasks per second. Only executing the original application, throughput peaks at about 1900 requests per second (Figure 9). Using Aaron as an underlying fault detection framework does not change the behavior: both versions reach the same throughput at the same point in time. Aaron does indeed not influence the throughput of the application.

At the right side of Figure 9 we show the fraction of checked executions. At start, the original system is loaded moderately. All requests are being processed using runtime checks. As soon as load increases at time $t_1$, the fraction of checked execution drops to zero. Once the request rate returns to its original level at time $t_2$, requests are processed using runtime checks again. The delay between time $t_2$ and the time at which the rate of checked executions rebounds again is explained by the large number of pending requests in the system. Once the backlog of requests is cleared, the fraction of checked execution is 1.0 again.

G. Dynamic Adaptation of Used Software Variants

We present the overhead for the different diversification methods in Figure 10. The overhead is normalized to the execution time of the version without runtime checks.

Although the absolute numbers are not a decisive factor in this work, their relationship to each other is important: Assertion, OverAllocate and in most cases also NullWrite
exhibit very low overhead. A typical application reads memory more often than it writes to it. Consequently, NullRead employs more runtime checks than NullWrite; thus it is more expensive.

The overhead of SWIFT and SWIFTCFC is significantly higher than of all other diversification methods. Actually it is even higher than reported in the literature [4]. This is due to our implementation: We do not use any of the optimizations presented in [4].

Developing highly-efficient diversification methods is orthogonal to Aaron, and not scope of this work. Instead, Aaron combines existing methods to increase the detection rate of latent errors.

The fraction at which certain alternatives are executed depends on the workload and the utilization of the system. The distribution for Wordcount is shown in Figure 11. As the utilization increases, Aaron adapts the runtime checking by choosing cheaper variants more frequently.

IV. RELATED WORK

Many approaches to fault detection demand significant additional resources from the system. They are expensive in terms of hardware and software. For example, the replicated state machine approach using Byzantine fault tolerant (BFT) protocols [19, 20] requires a minimum of $3F + 1$ machines to tolerate $F$ independent faults. A different approach is lockstep execution. Orchestra uses software-based lockstep execution to detect errors [21]. Similar to Aaron, it applies automatic software diversity. In contrast to redundant execution approaches like e.g. BFT protocols and Orchestra, Aaron does not necessitate additional hardware resources; it only exploits spare CPU cycles that are otherwise wasted.

Runtime checks are a more pragmatic approach to handle arbitrary failures: logging for replay [22, 23], monitoring [24, 25], data-flow integrity [7], hardware error detection [4], anomaly based checking [26, 10], input filtering [11, 27], and distributed assertions [12] to name a few. Runtime checks can be automatically applied by the compiler. In related work, the runtime checks are deployed on honeypots and systems where performance does not matter. In contrast, Aaron targets deployed systems with throughput critical applications. Aaron switches on checking only if it does not affect throughput.

If Aaron detects an error that it cannot tolerate, it drops the currently processed request. More sophisticated recovery approaches like Rx [9] or MicroReboot [28] could be combined with Aaron. Furthermore, our prototype does not provide strong sandboxing for request processing, yet. Again, it would be possible to integrate known sandboxing approaches like SafeDrive [29] or XFI [30] into Aaron.

Aaron was built with existing frameworks for distributed computing in mind. It could be ported to MapReduce [31], DryadLINQ [32] or similar approaches. In MapReduce applications, Aaron could exploit the not so uncommon performance heterogeneity [33].

There are several options to speedup runtime checking. The approach closest to ours is a feedback controller to limit the overhead of watchdogs in distributed embedded systems [34]. The feedback controller dynamically switches the watchdog on and off to adjust performance. Instead of embedded systems, we target data centers, which have different characteristics.

Parallelizing runtime checks is another approach to reduce the perceived performance overhead of runtime checks [35, 36, 17]. These approaches distribute load across additional CPU cores. In contrast to Aaron, they do not adapt to changing workloads dynamically.

Finally, sampling can be used to balance the false positive and false negative rate [10]. This system is complementary to Aaron.

V. FUTURE WORK

In the future, we plan to extend Aaron in 3 directions: First, using dynamic stack rewriting [17], Aaron could switch between different variants during the processing of a single task. This would enable Aaron to operate on applications that are not task-oriented. Second, we want to extend Aaron into the hardware domain by replicating
applications dynamically. Depending upon the load of the cluster environment, replication can be used to increase fault coverage even further. In contrast to traditional replication approaches, we want to downscale replication if necessitated by throughput demands. Third, we plan to explore new metrics to decide whether runtime checks should be employed. Processing costs fluctuate within one cluster and also across multiple different clusters. Scheduling runtime checks only if processing is cheap appears to be the next natural step for Aaron.

VI. CONCLUSION

For current server systems, the difference between power consumption at average load and peak load is small. Assuming an average load of 50%, using the remaining computing resources cost only about 14% (in our setup) in terms of power consumption.

Aaron exploits these spare cycles and schedules automatically diversified software variants to maximize fault coverage. Aaron’s on-demand error checking has no influence on the performance of the system, neither in terms of throughput nor in terms of responsiveness. The runtime overhead of Aaron is determined by the overheads of the different diversification methods only. Aaron enables the usage of automatic software diversification methods, even though the overheads of the methods themselves might be prohibitively high. It adapts failure coverage depending on the current load situation. This adaptivity enables the usage of Aaron in deployed systems without generating prohibitively high costs.

ACKNOWLEDGEMENTS

The authors thank Martin Nowack for valuable discussions and Figure 1. Parts of this research were funded in the context of the SRT-15 project by the European Commission under the Seventh Framework Program (FP7) with grant agreement number 257843.

REFERENCES

[16] U. Hölzle and L. A. Barroso, The Datacenter as


