ABSTRACT
Many programs have a fixed directed graph structure in the way they are processed. In particular, computer vision systems often employ this kind of pipe-and-filter structure. It is desirable to take advantage of the inherent parallelism in such a system. Additionally, such systems need to run in real-time for robotics applications. In such applications, robotic platforms must make time-critical decisions, and so any additional performance gain would be beneficial. To further improve on this, the platform may need to make the best decision it can by a given time, so that newer data can be processed. Thus, having a timeout that would return a good result may be better than operating on outdated information.

Categories and Subject Descriptors
B.2.1 [Arithmetic and Logic Structures]: Design Styles—parallel, pipeline; D.1 [Programming Techniques]: Concurrent Programming

General Terms
Algorithms, Design

Keywords
concurrent tree, pipeline

1. INTRODUCTION
Concurrency can be very useful for real-time systems, however it can often be difficult to manage [9, 11, 7, 10]. When dealing with robotic platforms [6], one may want to run data through multiple algorithms in order to confirm a result. These algorithms may need to run simultaneously in order to take advantage of the parallelism of multiple processes.

In addition, there should be some mechanism to get the best result available after some amount of time has passed.

BMP-RAP is based off of the pipe-and-filter paradigm [30], but also includes additional features such as thread and resource pooling, along with timeouts on finding the completed result. After a set amount of time has elapsed, the system guarantees to give the best result that it has at the time.

There are many problems, particularly in computer vision, which exhibit such structure naturally. For instance, most algorithms which employ a pyramid for analysis of an image in multiple scales have a structure which can be modeled in this way. In such a program, an image is fed into the system. It is then scaled successively by a factor, or alternatively some operation is performed on it several times with an increasing factor, and then the image is further operated on at each scale. At the end of the procedure, the results generated from processing each image is often collapsed down to a single set of results [26, 27].

1.1 Related Work
The pipe-and-filter model was used as a general basis for the architecture of BMP-RAP. This is because the model has very good properties when it comes to modularity, and because each filter can be considered as a single atomic operation [30]. There have been many groups which have worked on refining pipe-and-filter systems to make them optimal for various applications [28].

There has also been a lot of work in pipelining a single algorithm used for data analysis in order to take advantage of parallelism [28]. Unfortunately, these systems are limited because they do not take into account pooled resources which can be shared, including threads, and also are not designed for time-critical applications.

François proposed a hybrid generic architecture for combining the benefits of a pipe and filter model with shared resources [18]. This work is related, as it provides a method for parallelizing generic data streams through a pipe-and-filter model. Such a framework could be adapted to perform similar functionality to the system being proposed in this paper. However, this system was designed such that information
would continuously flow into the pipe and be output at the end "live". This is in contrast to the system being proposed where data flows in "real-time" once per time interval, and must be processed by a given deadline.

2. PROBLEM STATEMENT
Consider a program consisting of a tree of tasks \( t_0, t_1, ..., t_m \) where each task (Branch) \( t_i \) passes its result to all of its children. Each task can be considered as a single atomic operation on the data passed into it. All data passed into or out of a task is immutable. Multiple branches can feed into a single branch. Thus, the collection forms a directed acyclic graph which is to be traversed.

Every end node must either have the same type or be polymorphic to a single data type. At the end of an execution cycle, the data returned from each leaf node \( \delta_0, \delta_1, ..., \delta_k \) can be combined using some algorithm into a final result \( \delta \).

Each task can be executed on any thread \( T_i \in T_0, T_1, ..., T_n \). Regardless of completion status, the algorithm must be guaranteed to provide a result within \( \tau \) milliseconds. For non-time-critical applications, \( \tau \) can be set to \( \infty \).

2.1 Problem Analysis
Clearly, this system can become bottlenecked at any place where threads need to wait for a single node to complete execution before its children can be accessed. In the special case that the system is used on an entirely serial set of branches, the system becomes useless.

Additionally, execution may very drastically from choosing a different traversal method. Although depth-first traversal may arrive at the end faster for a given path, it may also not be the optimal strategy for executing parallel tasks.

3. EXAMPLE PROBLEMS
There are many algorithms which exhibit this directed acyclic graph structure. In this section, a few such algorithms are discussed.

3.1 SIFT Keypoint Detection
The Scale Invariant Feature Transform (SIFT) Keypoint Detector has a structure which could take advantage of parallelism. To identify keypoints, or stable features, of an image, a multi-tiered process is executed, which is described in detail in Lowe’s paper [26]. First, the algorithm resizes the image multiple times into \( m \) images. Each of these \( m \) images then has \( n \) Gaussian blurs applied to it. Then, the difference of each of the \( n \) Gaussian images is taken within each of the \( m \) scales. From here, the \( n \)-1 difference of Gaussian images are run through a series of tests. First, pixels of a given image which are identified as a local maxima within a 3x3 area are chosen. These maxima are then checked for cornerness using a Hessian, contrast with neighboring pixels, and also an interpolation step is used to find the more exact maxima or throw out outliers. These three steps can be performed in parallel and merged, in addition to being able to be performed in parallel to similar operations across the scale space.

Figure 1: Scale space of the SIFT Keypoint Detector [26]

3.2 Scale Space
The scale space is used in many other algorithms besides SIFT, such as in edge detection [27] and stereo or optical flow [4]. Thus, as a common operation which is used in many vision algorithms, while also being inherently parallel (since each image in the scale space must be operated on independently), makes such algorithms ideal for the proposed system.

3.3 Image Splitting
In a more general example, an image could be subdivided into multiple parts, each part fed into a different branch and operated on. This could be helpful during many fundamental operations, such as convolution, where only a small window is needed to be operated on at any given time.

4. SYSTEM OVERVIEW
In essence, the system is performing a tree traversal, where at each visited node, some operation is completed on it. For this system, there is a thread pool in order to mitigate the overhead of creating and destroying threads during runtime. A queue of nodes is created to list which branches are ready to be executed. The system is organized into three main classes: Tree, Branch, and Thread. An instance of Tree serves as the root node, and also stores references to the thread pool. Branches are the nodes of the tree, and are inherited from to create actual functionality.

For reasons of simplicity and rigor, all data passed from a node to its child is considered to be immutable. Data is passed to a branch’s children by returning it from said branch’s work function. The thread will then set each child’s input to the output of the branch. Finally, the thread adds each of the children to the node queue, and it starts operating on the next branch in the queue.

4.1 Using the System
To use the system, the user first creates an instance of Tree. When initialized, the user specifies the number of threads to be available in the pool, along with a function pointer to a
callback function, and then begins adding child nodes to the tree. To add children, the user simply creates instances of branches of different kinds to be pieced together for the tree. On each branch, the AddChild() function is called to add children to it. Children should only be added to Branches or the Tree before execution time. To activate, the user calls SetJob on the instance of the Tree.

4.2 User Implementation of a Branch
In order to implement a Branch, one would simply inherit from the Branch class. The user must then overload a single function, and has the option to overload one additional method.

```
virtual cv::Mat Process() const = 0;
```

Process() gets called when a thread has taken control of the branch. This function must be overloaded, and will do the actual work in the system. After the branch completes its work, it should return the data it wishes to pass to its children.

5. TREE, THREAD, AND BRANCH IMPLEMENTATION
The entire system was written in C++ because it is a common language that is widely used as of the writing of this paper. In order to support the development of BMP-RAP, several libraries were used: Boost [1], PThreads [3], and OpenCV 2.3.1 [2] In our future work we plan to experiment with the use of tasks as alternative to threads and specifically adopt Blaze Tasks [29].

As the three core structures of the framework, each serves a separate major function. This section describes the implementation details of each.

5.1 Branch
The following is a list of functions which are implemented as part of the Branch:

```
public:
// Called by a thread to execute the
// task (must be overridden)
virtual cv::Mat Process() const=0;
```

The Set, Reset, and Get functions simply wrap the functionality of the StringMap data structure, which is described in detail in a later section. STL vector and set structures are used for adding, accessing child branches. These can be used because we are working under our initial assumption that the structure of the graph does not change at execution time. None the less, they are protected by read/write locks in the event that a user does make a mistake and attempts to add a child during runtime.

IsReady checks that the StringMap contains all the names which were set using the AddArgument function. If any are missing, it returns false, otherwise, it returns true.

5.1.1 Terminal
For convenience, a modified implementation of Branch, Terminal, was created. This Branch behaves similarly to a normal branch, however it accepts a variable number of parameters. In order to accomplish this, the Set, Reset, IsReady, and Get functions are overridden. Instead of a StringMap, a Queue is used to store all the input arguments.
Terminals are intended to be used as a stopping point for the system, and one is called when the thread expires. Thus, there should only be a single Terminal instance per Tree. When a Thread is running, it does checks to see if a terminal is ready, which is determined by it either being full or by the timeout being triggered.

The user can subscribe to a terminal in order to get updates immediately upon their completion. To do this, the user shall implement a class which extends Terminal::TreeCallback, which is shown here:

```cpp
class TreeCallback
{
public:
    virtual void ProcessComplete(const cv::Mat& result, void* user);
}
```

Result is the result returned by the Terminal, and user is data which can be passed to the system upon subscribing for later use in the callback function. To support this functionality, the following three functions were implemented in Terminal:

```cpp
// Adds a subscriber to the list, along with // their associated user data.
void AddSubscriber(TreeCallback* subscriber, void* user);

// Removes the specified subscriber from the list.
bool RemoveSubscriber(TreeCallback* subscriber);

// Publish the result to all subscribers
void Publish(const cv::Mat& result);
```

In addition to this, a special function was created to combine data, rather than using Process(). This function was called Combine(). It was chosen to differentiate between a Branch which might just process data normally, and this Terminal which is merging information into a final result. The default combination method is to average all the images (unweighted). However, this Terminal could be extended to support other merge methods, such as a weighted average, AND operation, or other merging/filtering technique.

### 5.2 Thread

The thread is the workhorse of the system. Each one works asynchronously, attempting to get work from the BranchQueue, and adding more work to it after it has completed a given task. They work in a distributed manner without direction from a global arbiter. The following are some of the basic operations which can be called for managing the threads:

```cpp
void Run();
// Softly stops the thread from running
void Stop();
// Abruptly cancels thread execution (not // recommended except in extreme cases)
void Kill();
```

The real work is done in the ProcessThread() function, which is described in pseudocode as follows:

```cpp
loop while terminate_flag is not set
    loop while run_flag is set
        job = pop from BranchQueue
        if got a job
            exit loop
    end if
end loop

run Process() on job
write result to job's children
add all ready children to BranchQueue
end loop
```

5.3 Tree

The Tree class brings everything in the system together. Similar to branches, children can be added to it which are the first branches that will see the input image. The Tree also manages the thread pool, and provides the functionality to start and stop the system.

```cpp
// Initializes the Tree. timeout is // in milliseconds, default is 0 or // "no timeout".
bool Initialize(unsigned int numThreads, int timeout = 0);

// Removes all children of the Tree
void Clear();

// Sets the input image to be evaluated
void SetJob(const cv::Mat& job);

// Ends execution of the graph
void Stop();

// Adds a branch to be executed
void AddChild(Branch* child, const std::string& arg)
```

When SetJob is executed, each branch that is a child of the Tree gets its parameter set and is then added to the BranchQueue. Additionally, the start time is recorded for the purposes of the timeout system.

6. Lock-Free Data Structures

In order to increase parallelism of the system, it is desirable to use a queue for adding nodes which is Lock-Free. The necessary condition for lock free is to guarantee that always some function call completes in a finite number of steps [20, 25, 16, 31, 32, 12, 13, 5, 8, 15]. Another beneficial side-effect
of lock-freedom is that even if threads expire, the system will not deadlock. There is another, tougher restriction, Wait-Freedom [13, 15, 5, 14, 16], which requires that all calls will complete in a finite number of steps. For this system, it did not seem necessary to meet this condition for a couple reasons, mainly relating to the places where these data structures were used. More discussion is included in each structures’ description.

In order to meet the Lock-Free condition, we require an operation which has an infinite consensus number [20]. This operation, on most architectures, is or is similar to Compare And Swap. The prototype for the operation is as follows:

```c
CAS(pointer_to_value, old_value, new_value)
```

When executed, the processor atomically locks the memory bus, compares old_value to the value stored at pointer_to_value. If the two are the same, then it puts new_value at that location and returns true. Otherwise, it returns false.

For portability and reusability, all of the data structures implemented make use of templating. This has two benefits, first it makes the code reusable so that the data structure does not have to be implemented several times manually to support multiple data types. Secondly, templating was used instead of void pointers, which would also provide for reusability, because they are still strongly-typed. By being strongly-typed, the number of user-errors can be reduced since mistakes would be caught at compile-time.

Similar to the Standard Template Library (STL), data types used by all the data structures are assumed to have overloaded assignment operators [24], for example:

```c
LHS = RHS;
```

Which should always evaluate such that LHS will take on the value of RHS. Additionally, the data types must implement a default constructor.

### 6.1 Lock-Free Queue

The queue is used mainly for indicating which branches are available and ready for execution. It may be accessed by multiple threads simultaneously, where several threads may be committing ready branches to the back of the queue at the same time that other threads are attempting to get new work from the head. Rather than locking the entire queue for this system, a simple lock-free implementation was written to extract more parallelism out of the queue.

The following operations were defined for the queue (note that T denotes the type specified by the template argument):

```c
/// enqueues object at the tail of the list
void push_back(T object);

/// attempts to grab the head of the queue
/// if it fails or the list is empty, returns false and result is not set
/// if it succeeds, returns true and result is set to the head
bool pop_front(T& result);

/// will pop all nodes out of the list
void clear();

/// returns the size of the queue
unsigned int size();
```

A linked-list was chosen as the storage strategy for the queue in order to simplify the implementation. The queue uses a subclass as a helper for this linked-list, called Node, which is defined as follows:

```c
struct Node{
    T value;  /// The value of the entry
    Node* next; /// A pointer to the next entry
};
```

The queue itself has the following fields:

```c
/// The head of the list
Node* head;

/// A pointer to the next pointer of the tail
/// of the list
Node** tail;

/// The size of the list
unsigned int mSize;
```

#### 6.1.1 Enqueue (push_back)

The following is the pseudocode for the push_back operation:

```c
void push_back(T object){
    newnode = create_new_node();
    newnode.value = object;
    newnode.next = null;
    loop forever
        if CAS(tail.next, null, newnode) passes
            tail.next = newnode.next;
        loop
            sizestate = currentsize;
        until CAS(currentsize, sizestate, sizestate + 1)
            exit loop
    end if
end loop
```

The function first creates a new Node instance, settings its value to the inserted object. It then attempts to swap out the tail with this new Node. If it succeeds, it will then attempt to increment the size counter of the queue. Upon failure of either operation, it will continue to try again until it has completed.

This satisfies the lock-free condition, because one thread will always be able to complete the CAS operation, continuing its execution in a finite number of steps. The push linearizes as a completed operation as soon as tail.next points to the newnode’s next pointer.
6.1.2 Dequeue (pop_front)
The following is the pseudocode for the pop_front operation:

```c
boolean pop_front(T result reference) {
    oldhead = head;
    headnext = oldhead.next;
    if headnext is not null
        tempvalue = headnext.value;
        if CAS(head, oldhead, headnext) passes
            result = tempvalue;
            loop
                sizestate = currentsize;
                until CAS(currentsize, sizestate, sizestate - 1)
            return true;
        end if
    end if
    return false;
}
```

When dequeuing, a thread copies the head and then compare and swaps the head of the list with its next element. If the CAS passes, then it has successfully removed the head and can continue. Otherwise, the deque operation will return false. The operation will also return false if the queue is empty. Upon successful completion, the queue will also decrement the size of the list. Clearly, the size will not necessarily be faithfully up to date with the current state of the queue, however it is sufficient for this system.

As a desperate consumer, a thread would spin on the dequeue operation until there is work available, yielding after a fixed number of attempts in order to allow other threads to do work and potentially enqueue more nodes. Our design is not prone to corruption as a result of the ABA problem [7].

6.2 Lock-Free StringMap
The StringMap is used to store arguments passed to each branch, associating a string name with a templated object which is the actual value of the parameter. This map is implemented as a hash table which employs quadratic probing to deal with collisions. The key is always a standard C++ string, hashed using the FNV hash function. FNV Hash was chosen for its speed and simplicity [19].

```c
% the offset basis for 32-bit
hash = 2166136261
for each character in the string
    hash = hash XOR character
% the prime number for 32-bit
hash = hash * 16777619
end for each
return hash
```

The StringMap implements the following operations:

```c
/// Puts the value into the map at a
/// location specified by the key
/// returns true on success, false if
/// there is no place for the value
bool put(const std::string& key, T value);
```

```c
/// Gets the value stored at mapped location
/// key, and puts the value of it into value
/// returns true if such a matching key is
/// found, false otherwise
bool get(const std::string& key, T& value) const;
```

```c
/// Attempts to remove an object at the
/// location of the specified key. returns
/// false if there is nothing there, true
/// if something were successfully removed
bool remove(const std::string& key);
```

```c
/// will remove all entries from the map
void clear();
```

There is no resize operation implemented for this data structure. It is acceptable because we assume that the graph has a fixed structure when executed, thus a branch will know how many arguments it needs to take in by the time it is executed, so it can allocate an appropriately sized table before it begins running. Not implementing a resize function saved a lot of complexity that would be unnecessary for this application.

In order to implement this algorithm, a subclass called Node was created to encapsulate key and value information into a single structure at each location in the table. The following is the definition of the Node:

```c
struct Node {
    std::string key;
    T value;
}
```

6.2.1 Put
The following is the pseudocode for the put function:

```c
bool put(const std::string& key, T value) {
    hash = FNV_hash(key);
    probe_offset = 0;
    probe_factor = 0;

    node = create new node;
    node_key = key;
    node_value = value;

    location=0;
    loop
        location = hash+probe MOD capacity;
        old = table[location];
        increment probefactor;
        probe = 2^probefactor;
        until old_key equals key OR old_key is null
        until CAS(table[location], old, node)
    }
```
This function checks the key/value pair located at the ini-
tial hash value location modulo the table's capacity. If the 
keys are identical or if there is nothing stored there, then it 
attempts to update the value at that location. Upon failure, 
it will continue to probe the table for a location to store the 
value, incrementing by powers of 2.

6.2.2 Get
The following is the pseudocode for the get function:

```c++
bool get(const std::string& key, T& value) const
{
    hash = FNV_hash(key);
    probe_offset = 0;
    probe_factor = 0;

    location=0;
    capacitycheck=0;

    loop
        location = hash+probe MOD capacity;
        old = table[location];
        increment probefactor;
        probe = 2^probfactor;
        increment capacitycheck;
        if capacitycheck > 2*capacity
            return false;
        end if
    until old_key equals key

    value = old_key;
    return true;
}
```

Similarly to put and get, remove probes to find a matching 
key. Once it has found one, then it will attempt to remove 
the entry. If the removal completes, it frees the memory 
and returns true. If the CAS fails, however, then either an-
other thread deleted the entry or wrote over it. Rather than 
continuously attempting to remove, the operation concedes 
defeat. This behavior was chosen, because a removal should 
only occur if the graph is being cleared for new data, thus 
if new data were to be written in in the middle of removal, 
we don’t want to remove that information.

7. RESULTS
The system was run on an Intel Quad-Core i7 with Hyper-
threading, which had 8 logical cores. The machine also had 
8 GB of RAM. The machine ran Ubuntu 10.10. The tested 
algorithm was run using BMP-RAP with multiple numbers 
of threads and also run in serial, and their run times were 
compared.

The tested algorithm is not functional in a traditional sense, 
however does perform many operations typical of a computer 
vision system. The image was first resized, then a Gaussian 
blur was applied with 7 different kernels with sigma increas-
ing as a power of 2 for each level. The difference was taken 
between each adjacent blur, and then a Hough transform 
was run on each difference. In the end, all the images were 
averaged together to get the output image. Ten samples 
were taken under each condition to test its performance. In 
future work we plan to use advanced monitoring tools and 
techniques to gain a better understanding of the experimen-
tal results [17, 22, 23, 21].

8. ANALYSIS AND CONCLUSIONS
As expected, a single threaded run of BMP-RAP closely 
matches a separate serial implementation of the same al-
gorithm. The slight increase in runtime can likely be at-
tributed to the overhead of the system.

When 2 threads were used, there was a speedup of approxi-
ately 150 percent, which is very useful. Unfortunately, 
after this point, the speed actually reduced. Although on 
average 4, 6, and 8 threads performed better than a sin-
gle thread, it actually performed worse than 2 threads, and 
progressively worse as the number of threads increased. Ad-
ditionally, the run time appeared to get inconsistent when 
more threads were added, lending to the much larger stan-
dard deviation of run times.

This counters intuition, which implies there is likely some 
bottleneck in the implementation, where many of the threads 
are in contention. It is likely that when the 4-8 thread runs
performed well the scheduler happened to order the threads in an ideal pattern, and when it performed poorly they were timed just off enough to slow down the entire system.

It may be necessary to switch the BranchQueue to a wait-free implementation, as some threads may be starving.

9. FUTURE WORK

One desirable improvement would be to extend the system to be able to pass more than just cv::Mat’s between branches. Included with this, would be the ability to do type checking on such parameters to ensure that not only does the argument name match up, but also the data type.

One original plan was to make all arguments statically typed in order to make the system more fault tolerant and rigid. Unfortunately, it may be a limitation of the expressiveness of the language that prevents templating with multiple parameters from being a viable option in C++.

Currently, graph traversal is effectively a breadth-first traversal. Work could be done to experiment with depth first traversal. Additionally, a totally different scheduling approach could be used, where machine learning is used to interpret different paths to give optimal results under different time constraints.

10. ACKNOWLEDGMENTS

I would like to thank the hours and efforts that Dr. Damien Dechev has put into making this class possible for all of us.

11. REFERENCES

[1] Boost c++ libraries. http://www.boost.org/.
[2] OpenCV. http://www.opencv.org/.
[3] Posix.1c, threads extensions (ieee std 1003.1c-1995).
[4] T. K. B. D. Lucas. An iterative image registration technique with an application to stereo vision. *Proceedings of Imaging Understanding Workshop*, pages 121–130, 1981.
[5] A. Barrington, S. Feldman, and D. Dechev. A scalable multi-producer multi-consumer wait-free ring buffer. In *Proceedings of the 30th ACM/SIGAPP Symposium on Applied Computing*, Salamanca, Spain, April 2015.
[6] Damian Dechev and Bjarne Stroustrup. Reliable and efficient concurrent synchronization for embedded real-time software. In *Proceedings of the 3rd IEEE International Conference on Space Mission Challenges for Information Technology* (2009).
[7] D. Dechev. The aba problem in multicore data structures with collaborating operations. In *Proceedings of the 7th International Conference on Collaborative Computing: Networking, Applications and Worksharing*, Orlando, FL, October 2011.
[8] D. Dechev, P. LaBorde, and S. Feldman. Le/dc: Lockless containers and data concurrency: A novel nonblocking container library for multicore applications. *IEEE Access*, 1:428–435, September 2013.
[9] D. Dechev, R. Mahapatra, B. Stroustrup, and D. Wagner. C++ fast dynamic cast in autonomous space systems. In *Proceedings of 11th IEEE International Symposium on*.
Object/component/service-oriented Real-time Distributed Computing, Orlando, Florida, May 2008.

[10] D. Dechev, P. Pirkelbauer, N. Rouquette, and B. Stroustrup. Semantically enhanced containers for concurrent real-time systems. In Proceedings of the International Symposium and Workshop on Engineering of Computer Based Systems, pages 48–57, 04 2009.

[11] D. Dechev and B. Stroustrup. Scalable nonblocking concurrent objects for mission critical code. In Proceedings of 24th International Conference on Object-Oriented Programming Languages, and Applications, Orlando, Florida, October 2009.

[12] S. Feldman and D. Dechev. A wait-free multi-producer multi-consumer ring buffer. ACM SIGAPP Applied Computing Review, 15, October 2015.

[13] S. Feldman, P. LaBorde, and D. Dechev. Tervel: A unification of descriptor-based techniques for non-blocking programming. In Proceedings of the 15th IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, Samos, Greece, July 2015.

[14] S. Feldman, P. LaBorde, and D. Dechev. A wait-free multi-word compare-and-swap operation. International Journal of Parallel Programming, 43(4):572–596, February 2015.

[15] S. Feldman, P. LaBorde, D. Dechev, and C. M. level Arrays. Wait-free extensible hash maps. In Proceedings of the 13th IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, Samos, Greece, July 2013.

[16] S. Feldman, C. Valeraleon, and D. Dechev. An efficient wait-free vefctor. IEEE Transactional on Parallel and Distributed Systems, 27(3):654–667, May 2016.

[17] S. Feldman, D. Zhang, D. Dechev, and J. Brandt. Extending ldms to enable performance monitoring in multi-core applications. In Proceedings of the Monitoring and Analysis for High Performance Computing Systems Plus Applications, Chicago, IL, September 2015.

[18] A. R. J. François. A hybrid architectural style for distributed parallel processing of generic data streams. 26th International Conference on Software Engineering (ICSE’04), pages 367–376, May 2004.

[19] L. C. N. G. Fowler and K. P. Vo. The fnv non-cryptographic hash algorithm. March 2012.

[20] M. Herlihy and N. Shavit. The art of multiprocessor programming. 2008.

[21] R. Izadpanah, B. A. Allan, D. Dechev, and J. Brandt. Production application performance data streaming for system monitoring. ACM Transactions on Modeling and Performance Evaluation of Computing System, March 2019.

[22] R. Izadpanah, S. Feldman, and D. Dechev. A methodology for performance analysis of non-blocking algorithms using hardware and software metrics. In Proceedings of the 19th IEEE International Symposium on Object/component/service-oriented Real-time Distributed Computing, York, UK, May 2016.

[23] R. Izadpanah, N. Naksinehboon, J. Brandt, A. Gentile, and D. Dechev. Integrating low-latency analysis into hpc system monitoring. In Proceedings of the 47th International Conference on Parallel Processing, Eugene, OR, August 2018.

[24] JTC1/SC22/WG21. Iso/iec 14882:2003. pages 465–543, 2003.

[25] P. LaBorde, S. Feldman, and D. Dechev. A wait-free hash map. International Journal of Parallel Programming, 45:421–448, 2017.

[26] D. G. Lowe. Distinctive image features from scale-invariant keypoints. Int. J. Comput. Vision, 60(2):91–110, November 2004.

[27] P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. IEEE Trans. Pattern Anal. Mach. Intell., 12(7):629–639, July 1990.

[28] J. Philippes and B. Rumpe. Refinement of pipe-and-filter architectures. FM ’99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems, 1:96–115, 1999.

[29] P. Pirkelbauer, A. Wilson, C. Peterson, and D. Dechev. Blaze tasks: A framework for computing parallel reductions over tasks. ACM Transactions on Architecture and Code Optimization, 15(4), January 2019.

[30] M. Shaw and D. Garlan. Software architecture - perspectives on an emerging discipline.

[31] D. Zhang and D. Dechev. A lock-free priority queue design based on multi-dimensional linked lists. IEEE Transactional on Parallel and Distributed Systems, 27(3):613–626, April 2016.

[32] D. Zhang and D. Dechev. Lock-free transactions without rollbacks for linked data structures. In Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, Pacific Grove, CA, July 2016.