Making Break-ups Less Painful: Source-level Support for Transforming Legacy Software into a Network of Tasks

Nik Sultana
University of Pennsylvania

Achala Rao
University of Pennsylvania

Zihao Jin
Tsinghua University

Pardis Pashakhanloo
University of Pennsylvania

Henry Zhu
University of Pennsylvania

Ke Zhong
Shanghai Jiao Tong University

Boon Thau Loo
University of Pennsylvania

ABSTRACT

"Breaking up" software into a dataflow network of tasks can improve availability and performance by exploiting the flexibility of the resulting graph, more granular resource use, hardware concurrency and modern interconnects. Decomposing legacy systems in this manner is difficult and ad hoc however, raising such challenges as weaker consistency and potential data races. Thus it is difficult to build on battle-tested legacy systems.

We propose a paradigm and supporting tools for developers to recognize task-level modularity opportunities in software. We use the Apache web server as an example of legacy software to test our ideas. This is a stepping stone towards realizing a vision where automated decision-support tools assist in the decomposition of systems to improve the reuse of components, meet performance targets or exploit new hardware devices and topologies.

CCS CONCEPTS

• **Computer systems organization** → **Maintainability and maintenance**; • **Software and its engineering** → **Extra-functional properties**; **Software post-development issues**;

KEYWORDS

program analysis; program transformation; distributed systems

ACM Reference Format:
Nik Sultana, Achala Rao, Zihao Jin, Pardis Pashakhanloo, Henry Zhu, Ke Zhong, and Boon Thau Loo. 2018. Making Break-ups Less Painful: Source-level Support for Transforming Legacy Software into a Network of Tasks. In *FEAST '18: 2018 Workshop on Forming an Ecosystem Around Software Transformation, Oct. 15, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/XXXXXX.XXXXXX

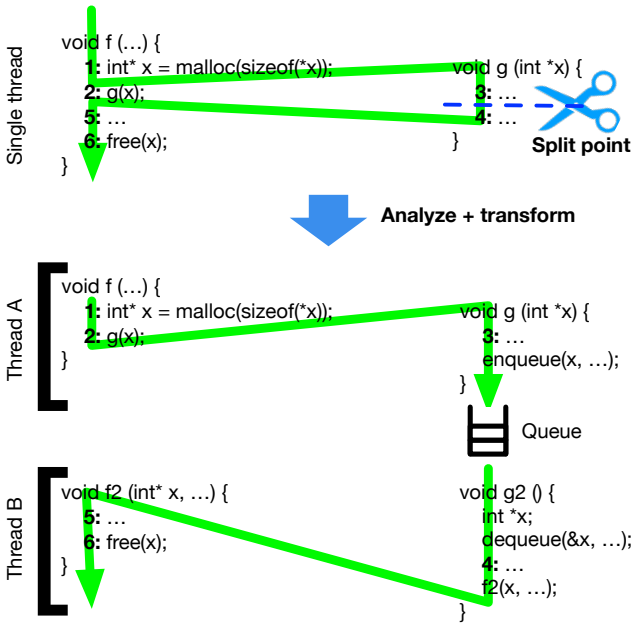


Figure 1: Transforming the subroutine call/return control-flow paradigm to one based on enqueue/dequeue over channels between threads. We must analyse programs to ensure that sufficient context is passed from one thread to the next, that the transformation will not produce name clashes or type errors, and to avoid introducing data races between threads (e.g., if one thread frees a resource before another thread finishes using it).

1 INTRODUCTION

The benefits of modular system design have long been appreciated in engineering both to improve the complexity of designing a system by decomposing it into simpler parts, and to improve the system's operation through subsystems operating in parallel. In programming, the concept of component-based software traces its roots back decades to proposals for component-based operating systems [6], databases [2], networking [11], and service-oriented architectures for web services [16].

In recent years, fine-grained component-based programming has resurged with the advent of microservices-based cloud services partly as a reaction to the unnecessary duplication of features when using system- or OS-level virtualization. The increased interest in microservices was also complemented by improved architectural support for parallelism in off-the-shelf multicore devices, ongoing

FEAST '18, October 15, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5997-9/18/10.
https://doi.org/10.1145/XXXXXX.XXXXXX

research into manycore designs [1], and more modern interconnects within and across cores, sockets, machines and racks of machines [15].

Legacy code usually cannot take advantage of these improvements without extensive modifications. One can make use of component-based frameworks by coarse-grained blackbox encapsulation of legacy code as outsize components, as often done in NFV [19] where third-party vendors package network functions as reusable components for composition. However making fullest use of component-based frameworks (e.g. as done in Click [11]) requires either a clean-slate rewrite of existing code into components (e.g., as for TCP [13]), or painstaking manual work in retrofitting legacy code into components (e.g. Scalanytics [8]).

Decomposing or splitting legacy code into components is a challenging problem that different communities have explored for some time, as outlined in §5. Some of the challenges include determining the right granularity of decomposition, and whether decomposed modules can interact with each other efficiently. This can involve a combination of static and runtime analysis of code (§5). This line of work has so far not resulted in significant wide-spread use, largely due to the complexity of software systems which makes it difficult to devise general automatic methods.

Despite the above challenges, we argue that such decomposition of legacy software can bring several practical benefits. First, one can debloat complex software by dynamically recomposing it (i.e. assemble only what you need) to meet application requirements. Second, one can better mitigate resource misuse (e.g., leaks) and abuse (e.g., denial-of-service attacks) due to finely-granular sharing of resources [4]. Third, by organizing software into a dataflow network we can take better advantage of more hardware cores available on modern architectures. Finally, by reducing more application-level behavior into network-level behavior we can scale applications as we would scale networks: for example through increased redundancy, more careful routing, and service differentiation. As we demonstrate in our Apache example later, this decomposition can be used to have applications meet service-level objectives to improve overall availability [20].

In this paper, we present the Chopflow tool that aims to make it easier to modify legacy software when splitting it up into finer-grained communicating elements. Chopflow is generally applicable to any userspace software, though our initial focus is on network-centric applications.

In a nutshell, we view splitting as introducing or reorganizing the thread tasking of a program, and setting up channels over which the legacy and new threads communicate, as illustrated in Fig. 1. Unlike prior approaches aiming for full automation, our approach is designed to involve the human in-the-loop, where analysis tools assist programmers to evaluate where and how to split in order to meet their performance objectives.

Our approach involves static analysis of software to manage the combinatorial exploration on behalf of the human, but leaves the final partition decision to the domain experts. We argue that this approach is more likely to work across a wide-range of complex software, where full automation is not achievable, and facilitate better control of application-level behavior (such as managing sessions or user quotas) that a programmer is more likely to understand and debug successfully.

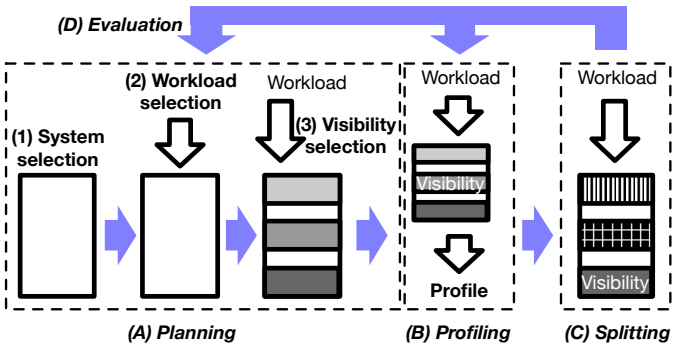


Figure 2: Steps taken in splitting an application and finding feasible split points.

We make the following contributions:

- **Chopflow tool.** We are developing the Chopflow tool, that analyzes application source code then provides the programmer with advice on what auxiliary changes to make when splitting. As we note later in the paper, this is nontrivial in legacy and portable multithreaded software that evolved their own runtime platforms, such as Apache.
- **Case study and preliminary evaluation.** We give an example of using Chopflow on the Apache web server, a non-trivial network-based application whose performance is mission-critical to many. We describe how we apply our tool to Apache when splitting it to mitigate a low-volume denial-of-service attack.

Based on our initial findings we describe some of the challenges and opportunities ahead.

2 SPLITTING METHODOLOGY

Figure 2 shows the steps we take when splitting up an application. We proceed in four phases: (A) Planning, (B) Profiling, (C) Splitting, and (D) Evaluation.

In the *Planning* phase (A), we pick the application to be split (1). The input to our methodology consists of the application source code (including any libraries that it uses, if available).

We then find workloads that expose interesting behavior (2). For example, as we will later show in the Apache use case, the workload is selected to identify code-splitting strategies to mitigate against attacks.

Finally in this phase, we determine how much of the system to observe (3). Given that the source code can be large, the cut-off to visibility abstracts away non-essential details to focus our attention on (parts of) the application and supporting libraries. This steps trades off visibility and analysis time. The larger the system we analyse, the more information we might have to sift through, a lot of which might be redundant.

In the *Profiling* phase (B) we run the application using the workload in order to generate a runtime profile. We compare the profiles of the software when running different workloads (e.g., normal and abnormal workloads), to localise the workload's symptoms in the source code. This gives us clues about where to split.

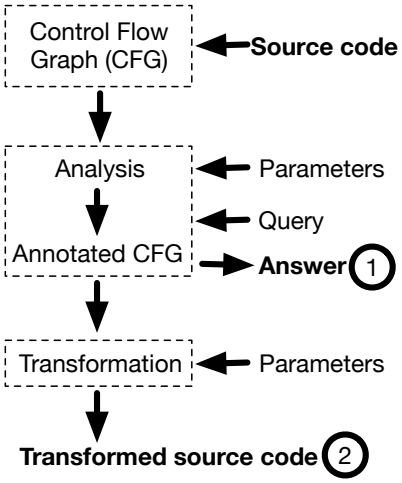


Figure 3: Chopflow processes source code to produce two kinds of outputs: (1) Answers to queries about the source code, and (2) transformed source code.

Next in the *Splitting* phase (C) we use the information gathered during profiling to consider where to split the software. We split before unwanted behavior is triggered, and rearchitecture the software to avoid the unwanted behavior. The key activities when splitting consist of: 1) splitting sequences of statements (across or within functions) into threads; 2) setting up queues between these threads, and starting these threads up at a suitable point in the system startup (and "joining" with these threads at a suitable point at system shutdown); 3) replacing sequential advancement or function call/return with enqueuing/dequeuing.

Our work is primarily intended to help the programmer discover opportunities for carrying out the first and third of these activities, and in future, provide support for automated program rewrites for the second step. This paper describes the design of a tool to provide this support.

Finally, in the *Evaluation* phase (D) we rerun the workloads to measure the performance implications of the split. The process might iterate several times, for instance, if we find that more visibility into the system is needed, or if the split did not achieve the desired behavior change.

3 CHOPFLOW

Chopflow is the decision-support tool we are building for software splitting. It works by abstracting source-code into a graph that models the program's control flow. This graph can be annotated with further information supplied by the user (e.g., which entry point they want to set, and which functions are related to the allocation and freeing of resources). The resulting model is then used to (1) answer queries made by the user (e.g., would there be a leak or data race if a program were split at a given line), or (2) transform the software (either by inserting annotations, or by splitting one thread into two), as shown in Fig. 3.

Chopflow is designed to be easily extensible to include the program's API in its model. Unlike previous approaches, we allow the program model to be enriched with user-supplied details on *characterising functions*: for example, which functions terminate the program, which functions are related to enqueuing and dequeuing, and which functions implement resource acquisition and release. This improves our reasoning about the program, even if the program does not directly use a well-known API (e.g., POSIX). Furthermore, Chopflow is able to detect and use so-called *junctions* between threads: these are locations where one thread enqueues work for a later thread to process. Junctions are used to analyse the continuity of processing of data between threads.

When starting up, Chopflow performs initial checks on the program: it ensures that the program can be parsed (by leveraging Clang/LLVM [12]), and, if asked about a split, it checks that the split point intersects the code paths we care about. If either of these are false, then the analysis does not proceed further.

After the initial checks, we use knowledge of the characterising functions to 1) compute data flows to determine what other code needs to be "moved forward" from one scope to another to preserve the relative temporal order: for instance, cleanup on data cannot occur in the original thread if that data is still being processed by a downstream thread. 2) we check whether the thread can indeed hand-off the computation: this cannot be done for example if the thread is alternating between processing two connections that influence each other (e.g., as in a proxy). In this case, it is likely that the split needs to hand-off *both* connections in order to preserve dependencies.

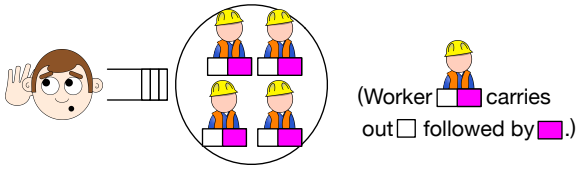
Our current implementation of Chopflow only performs analyses for question-answering. We are currently extending it with additional heuristic analyses, and to perform source-code transformations.

4 USE-CASE: SPLITTING APACHE

We use the Apache web server as our driving example to illustrate in more detail the steps to identify and implement possible split points in a software. Apache has been developed over more than 20 years and is a very widely-used system.[1] It supports multiple protocols including multiple versions of HTTP, and legacy features that are retained for backward compatibility. Apache works on a variety of operating systems and their compilers and TCP implementations, and tries to address a wide variety of needs over a long period of existence. These qualities make it a good example of a legacy network-facing server application.

Apache uses an abstraction API called "multi-processing modules" (MPMs) to organize its processing, and the most mature multi-threaded MPM is the *worker* MPM which dedicates a thread for the entire duration of a connection.

The worker MPM is sketched below: a listener thread queues connection descriptors to be serviced by a worker thread drawn from a worker pool. Workers carry out a sequence of computations on a connection; we will split this sequence into cooperating threads.



[1] https://news.netcraft.com/archives/category/web-server-survey/

The splits we describe in the next section involve building deeper pipelines than what the standard MPMs provide.

Objective for split. In this case study, our goal when splitting Apache is to address its vulnerability to low-volume denial-of-service attacks, and in this paper we focus on SlowLoris.[2] Our strategy to mitigate such DoS attacks involves reducing the application into a dataflow network to utilize resources better and contain the effect that clients have on one-another.

In the rest of the section, we describe by example the phases described in Fig. 2.

Phases A and B: Preparation and Profiling

Workload selection. We used different workloads to understand Apache's behavior in various conditions: 1) downloading a small file; 2) downloading a large file; 3) Apache Bench for latency and throughput measurements; 4) SlowLoris attack script. These workloads are used to compare normal from abnormal Apache behavior, hence allowing us to compare profiles to identify code hotspots or slowdowns caused by attacks. The thinking here is similar to experiment design, and the workloads need to be repeatable.

Visibility selection. We initially focused our attention on the Apache system alone: i.e., the core server and all the modules, ignoring other parts of the system. We later decided to also profile the Apache Portable Runtime (APR) to understand the function calls made by Apache into the APR since some calls were related to queue and resource management, and were therefore important to us.

There are many details of Apache that were removed from our visibility. Apache's operation is inherently complex because it interacts with various other features of a system, such as the OS' access control system, file system, language interpreters, back-end database, etc.

Profiling. For each workload we measured the amount of time that was spent in different functions. This provided us with an initial approximation about which part of Apache was being stalled by SlowLoris.

Phase C: Splitting

Splitting is a hypothesis-guided activity based on the semantic understanding acquired about the system during the previous steps.

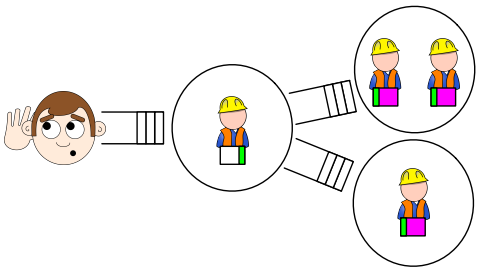
From profiling we have a rough idea of *by when* to split at the function call granularity, but one cannot simply split at any line of code preceding the point where "good" and "bad" workloads' profiles diverge. We need to find a *feasible* split point at or before a major divergence point. We define feasibility to mean that 1) the majority of execution paths pass through it, 2) little code needs to be changed elsewhere to accommodate the split, 3) the split does not break any dependencies in the code. The combination of these qualities allows us to safely implement the sketch from Fig. 1.

The Chopflow tool establishes feasibility heuristically by statically analysing the program's source code. As described in §3, we are able to compensate for static analysis' lack of precision by using knowledge of characterising functions to annotate our model of the program.

[2]https://en.wikipedia.org/wiki/Slowloris_(computer_security)

We picked a point at which to split the worker thread just after the connection has been initiated and then established by the listener. To mitigate SlowLoris, which is an application-level attack, the processing in the first worker pool must not reach a stage that is exploited by SlowLoris.

After splitting at this point, we end up with two kinds of workers organized along a pipeline. The listener thread queues jobs for a first worker pool which after doing some processing hands over to a second worker pool. We then partition the second worker pool into two groups to load-balance depending on the client's IP address. Note that we do not alter the number of threads, but rather *partition* them into different roles:



The small green boxes along the white and pink boxes we saw previously indicate that splitting introduces new code to send and receive values over queues. This design is intended to make it harder for a few bad apples to spoil it for all clients: the effects of clients are contained by partitioning the threads that clients can influence using SlowLoris.

The resulting change to the worker MPM is in the order of 100 lines. From the original worker thread we gather descriptors into a record and push them into a queue:

```
+struct conn_queue_entry * entry =
+    apr_pcalloc(p, sizeof *entry);
+entry->conn = current_conn;
...
+rv=apr_queue_trypush(destination_q,entry);
-ap_process_connection(current_conn, sock);
-ap_lingering_close(current_conn);
```

and we remove subsequent clean-up to avoid a race with the thread that will dequeue the connection record. Then in the second-stage worker we dequeue, update some internal descriptor information (to refer to the current thread), do the processing, and carry out the clean-up. We use APR facilities for threads and queues in our patch; mixing concurrency frameworks would be asking for trouble. This supports our view that it is best to keep a human in the loop to make splitting decisions.

We continued generalizing our code to parametrically split the backend worker pool into N disjoint groups, and implemented work-stealing by workers in idle groups when other groups' queues exceed a certain threshold.

Phase D: Evaluation

In this phase we check Apache's availability when under a SlowLoris attack. We compare the release version of Apache version 2.4.26 against a partitioned copy of 2.4.26 that was modified as described

above. We used the default Apache configuration for both experiments, and our split Apache uses the same number of threads as the original.

The experiments consisted of a machine running Apache and two client machines: one running Apache Bench and the other running a SlowLoris attack against the server. The machines were connected over 1 Gbps links via a switch.

The results are shown below, and indicate that the split software has improved availability, as was intended by the splitting. This was achieved because the resources (worker pool) were partitioned, which also effectively partitioned the clients who can indirectly allocate work to each pool. This partitioning enables us to contain the abuse that one set can carry out on another.

		Average latency ± std.dev. (ms)		
Attack	Split	Connect	Total	KReq/s
N	N	0 ± 0.2	9 ± 0.9	5.2
	Y	0 ± 0.3	11 ± 1.6	4.5
Y	N	0 ± 0.2	1099 ± 4818.0	0.039
	Y	0 ± 0.3	11 ± 1.3	4.5

The table shows latency and throughput, both measured by Apache Bench applied to the stock and partitioned versions of Apache 2.4.26. The experiment was done twice: first running Apache Bench in normal conditions, and second running it during a SlowLoris attack.

5 RELATED WORK

Software decomposition has been researched for different objectives in different programming languages [3, 5, 14, 17]: 1) security through compartmentalization [9, 18], 2) offloading and "cyber foraging" [25], 3) automatic parallelization [24], and 4) scalability research in networking [4, 21, 22] and more generally [7, 23] to better utilize hardware cores. Our approach is also related to the general idea of profile-guided optimization [10], but seeks to be application-oriented (as opposed to application-transparent as in just-in-time compilation, for example). From the related work, we make the following observations about software splitting:

- The most successful automation in prior work is done to achieve low-level objectives (e.g., keeping more cores busy) rather than application-level objectives (e.g., handling part of a request to stymie a denial-of-service attack).
- Part of the difficulty when partitioning legacy software consists of recognizing the concurrency (and auxiliary functions, such as locking) that already exists in the software.
- Handling state is crucial, and related work describes the management of different kinds of state [21, 22], but leaves unanswered the question of how to support the isolation of that state.

6 CONCLUSION

Our approach to software decomposition and parallelisation acknowledges the difficulties identified in earlier work and seeks to improve the problem's tractability by implementing a suite of analyses to help the programmer discern possible mappings of a program into topologies of tasks.

We believe that this approach offers the programmer more control over how the program is decomposed, and enables reuse of facilities that an application already uses, as in the Apache use-case (§4). This makes the split code more homogeneous and thus simpler.

In ongoing work we are i) splitting more software to develop additional use-cases, ii) implementing more analyses to support the automated splitting of subroutines into a pipeline of threads, iii) automating this transformation, and iv) exploring how to adapt our thread-based approach to work with event-based systems (such as Apache's Event MPM, and Nginx).

ACKNOWLEDGMENTS

We thank Bob DiMaiolo for early assistance with this project, and John Frommeyer for systems support. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contracts No. HR0011-17-C-0047 and No. HR0011-16-C-0056, and NSF grants CNS-1513679 and CNS-1563873. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA or NSF.

REFERENCES

[1] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. 2016. OpenPiton: An Open Source Manycore Research Framework. *SIGPLAN Not.* 51, 4 (March 2016), 217–232.

[2] Don Batory and Sean O'Malley. 1992. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Trans. Softw. Eng. Methodol.* 1, 4 (Oct. 1992), 355–398.

[3] István Bozó, Viktoria Fordós, Zoltán Horvath, Melinda Tóth, Dániel Horpácsi, Tamás Kozsik, Judit Köszegi, Adam Barwell, Christopher Brown, and Kevin Hammond. 2014. Discovering Parallel Pattern Candidates in Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang (Erlang '14)*. ACM, 13–23.

[4] Ang Chen, Akshay Sriraman, Tavish Vaidya, Yuankai Zhang, Andreas Haeberlen, Boon Thau Loo, Linh Thi Xuan Phan, Micah Sherr, Clay Shields, and Wenchao Zhou. 2016. Dispersing Asymmetric DDoS Attacks with SplitStack. In *ACM Workshop on Hot Topics in Networks (HotNets)*.

[5] David del Rio Astorga, Manuel F. Dolz, Luis Miguel Sanchez, and J. Daniel García. 2016. Discovering Pipeline Parallel Patterns in Sequential Legacy C++ Codes. In *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'16)*. ACM, 11–19.

[6] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. 1995. Exokernel: An Operating System Architecture for Application-level Resource Management. In *ACM Symposium on Operating Systems Principles*. 251–266.

[7] S. M. Farhad, Yousun Ko, Bernd Burgstaller, and Bernhard Scholz. 2012. Profile-guided Deployment of Stream Programs on Multicores. *SIGPLAN Not.* 47, 5 (June 2012), 79–88.

[8] Harjot Gill, Dong Lin, Xianglong Han, Cam Nguyen, Tanveer Gill, and Boon Thau Loo. 2013. Scalanytics: A Declarative Multi-core Platform for Scalable Composable Traffic Analytics. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*. ACM, 61–72.

[9] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. 2015. Clean Application Compartmentalization with SOAAP. In *ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, 1016–1031.

[10] Erik Johansson and Sven-Olof Nyström. 2000. Profile-guided Optimization Across Process Boundaries. *SIGPLAN Not.* 35, 7 (Jan. 2000), 23–31.

[11] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The Click Modular Router. *ACM Trans. Comput. Syst.* 18, 3 (Aug. 2000), 263–297.

[12] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[13] Rafael Laufer, Massimo Gallo, Diego Perino, and Anandatirtha Nandugudi. 2016. CliMB: Enabling Network Function Composition with Click Middleboxes. *SIGCOMM Comput. Commun. Rev.* 46, 4 (Dec. 2016), 17–22.

[14] Mihai T. Lazarescu and Luciano Lavagno. 2015. Interactive Trace-Based Analysis Toolset for Manual Parallelization of C Programs. *ACM Trans. Embed. Comput. Syst.* 14, 1, Article 13 (Jan. 2015), 20 pages.

[15] Sergey Legtchenko, Nicholas Chen, Daniel Cletheroe, Antony Rowstron, Hugh Williams, and Xiaohan Zhao. 2016. XFabric: A Reconfigurable In-Rack Network for Rack-Scale Computers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, 15–29.

[16] Angel Lagares Lemos, Florian Daniel, and Boualem Benatallah. 2015. Web Service Composition: A Survey of Techniques and Tools. *ACM Comput. Surv.* 48, 3, Article 33 (Dec. 2015), 41 pages.

[17] Huiqing Li and Simon Thompson. 2013. Multicore Profiling for Erlang Programs Using Percept2. In *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang (Erlang '13)*. ACM, 33–42.

[18] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *USENIX Annual Technical Conference*. USENIX Association, 285–298.

[19] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Berkeley, CA, USA, 459–473. http://dl.acm.org/citation.cfm?id=2616448.2616491

[20] Jeffrey C. Mogul, Rebecca Isaacs, and Brent Welch. 2017. Thinking about Availability in Large Service Infrastructures. In *Proc. HotOS XVI*.

[21] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. 2013. Escape Capsule: Explicit State is Robust and Scalable. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems (HotOS'13)*. USENIX Association, 10–10.

[22] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. 2013. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, 227–240.

[23] Ram Rangan, Neil Vachharajani, Guilherme Ottoni, and David I. August. 2008. Performance Scalability of Decoupled Software Pipelining. *ACM Trans. Archit. Code Optim.* 5, 2, Article 8 (Sept. 2008), 25 pages.

[24] Margo Seltzer. 2015. Automatically Scalable Computation. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15)*. ACM, 283–283.

[25] M. Sharifi, S. Kafaie, and O. Kashefi. 2012. A Survey and Taxonomy of Cyber Foraging of Mobile Devices. *IEEE Communications Surveys Tutorials* 14, 4 (April 2012), 1232–1243.