SEEC-AP: Self-Aware Software Architecture Patterns

Jim Holt[1,2], and Henry Hoffmann[2]
[1]Freescale, Networking Systems Division, [2]MIT Computer Science & Artificial Intelligence Laboratory
jim.holt@freescale.com; {jholt,hank}@csail.mit.edu

***Abstract* - The SEEC model provides methodology and infrastructure for creating self-aware systems that dynamically self-optimize to meet performance, power, and accuracy goals. In general, the ability of software implementations to achieve such goals is largely governed by principles of software architecture; and, software architecture patterns guide application developers towards robust and reusable implementations by capturing and codifying architectural best practices. We explore combining SEEC with software architecture patterns to achieve self-optimization while promoting both separation of concerns and reuse. A UML model for SEEC architecture patterns is presented (SEEC-AP). A prototype implementation was created and initial experiments were conducted with two streaming sensor applications. Results suggest that SEEC-AP can help accomplish effective self-optimizing applications while reducing the burden on application developers.**

I. INTRODUCTION

Manually optimizing software can be complex and time consuming, especially (i) when optimizing for multiple metrics such as power, performance, and accuracy, or (ii) when applications are intended to serve a long lifetime in varied deployment environments. For these scenarios the number of potential configurations is simply more than an application developer can reasonably evaluate [1]. Furthermore, many such applications may work against large datasets, may be long-running, or may exhibit significantly different execution phases over time. And, an optimal configuration for a given computing environment, execution phase, or dataset may not be the best choice for a different computing environment, execution phase, or dataset. These applications can benefit from self-optimizing code that adjusts to meet goals under varying conditions.

This paper proposes a new approach to using software architecture patterns for self-optimizing software. The approach, called SEEC-AP, involves encapsulating SEEC self-awareness capabilities [2] within parallel software architecture patterns [3]. SEEC-AP allows pattern implementations to provide the mechanisms and policy for interacting with the SEEC model, based upon the semantics of the pattern itself. In turn, pattern implementations can be re-used by application developers who instead focus their efforts on crafting application code, and on providing adaptations (e.g., "knobs") that collaborate with SEEC to achieve dynamic self-optimization.

Experiments with two pipelined benchmarks show that SEEC provided power reductions between 9% and 73% versus baseline configurations, and reductions in standard deviation of stage processing times between 68% and 89%. In one case SEEC increased power slightly (8%), yet pipeline stage processing time standard deviation improved by 67%.

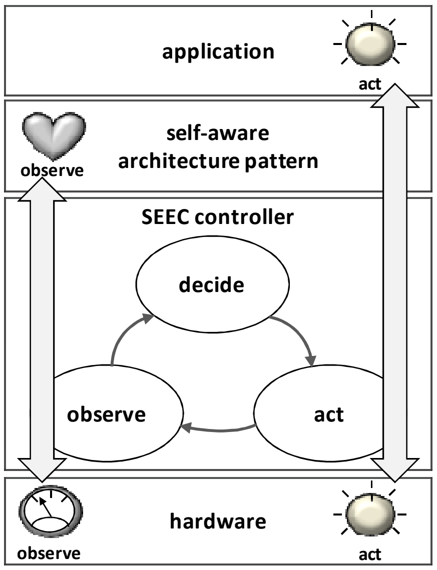


Fig. 1. SEEC mapped to software architecture

The SEEC model includes an *observe-decide-act* loop (Fig. 1). With SEEC, applications use one interface to specify goals and adaptations (for example, an image analysis application needs to process one image per second, or it can offer to increase or decrease the number of threads used for processing), while system software and hardware use a separate interface to specify actions (for example, changing core frequency, or allocation of threads to cores). This is complemented by a runtime decision system that determines how to use the available actions to meet goals while reducing costs.

With SEEC, applications explicitly state their goals and other system components measure whether goals are being met. SEEC uses the Application Heartbeats framework [4] to specify application goals and mark progress. The API's key abstraction is the heartbeat; applications use a function to emit heartbeats at important intervals, while additional API calls specify goals in terms of the heartrate. SEEC currently supports three different application-specified goals: performance, power, and accuracy. Performance is specified as a target heartrate or a target latency between specially tagged heartbeats. Power and energy goals can be specified as target average power for a given heartrate or as a target energy between tagged heartbeats. Accuracy goals are specified as distortion measured over some set of heartbeats. A second API allows a controller in the system to observe the current value of these metrics. Various approaches have been used to specify adaptations available to the controller [2, 4, 5].

The SEEC model can be mapped to software architecture patterns, providing clarity and separation of concerns, while also promoting reuse across many applications within many application domains. The key insights are as follows:

- For a given software architecture pattern, the "hooks" required to effectively utilize the SEEC model can be generalized
- Given this generalization, the implementation details of interacting with SEEC can be encapsulated in the pattern implementation
- SEEC adaptations (e.g., "actions" or "knobs") can also be generalized, allowing application-specific adaptations to be provided by the developer in a simple and clear way, while control and sequencing of adaptation can be provided by the pattern implementation, exploiting the natural semantics of the pattern
- By creating a set of composable architecture patterns, the proper coordination of multiple control loops can also be encapsulated, while still providing flexible architecture strategies to the programmer

Capitalizing on these insights, the contributions of this paper include:

- A UML-based model for composable SEEC software architecture patterns (SEEC-AP)
- A prototype SEEC-AP implementation of two parallel software architecture patterns
- Initial experimentation using SEEC-AP with two different streaming applications.

II. BACKGROUND

Several successful examples of self-optimizing libraries or frameworks exist. These tend to be specialized towards specific application domains or algorithm classes (e.g., FFTW, ATLAS, and SPIRAL, among others [1, 6-8]), or provide infrastructure that can be used to build self-optimizing applications (for example Application Heartbeats, Dynamic Knobs and the Self-Aware Computing (SEEC) model [2, 4, 5]). While quite effective when properly deployed, these tools provide limited guidance to software developers regarding how to structure their application to achieve best results. On the other hand, architecture is central to ensuring that an application can meet both functional and non-functional requirements [9]. Hence, a well-chosen software architecture can help separate concerns of meeting application requirements from concerns of employing self-optimization techniques.

Creation of monitoring, modeling, and adaptation mechanisms from scratch for each new application is expensive, further motivating reuse enabled by software architecture principles [9-11]. Thus, many software engineering researchers have advocated the importance of software architecture in *autonomic computing* [12-17]. Yet, architectural guidance provided by autonomic computing speaks little to the internal software architecture of the application, focusing instead on architecture relating the application to the larger autonomic system (itself composed of many separate, unrelated applications). Most proposed autonomic architectures define an explicit, first-class control-loop that applications must interact with [10, 15, 17-20]. Proponents argue that it is good engineering practice to aim for a single control loop, and when this is not possible the interactions of control loops should be carefully managed, as in the case of decentralized control (an important concern in future manycore systems) [21]. Furthermore, an explicit control loop can deal with sequencing and control issues, such as coordinating when the system can be safely reconfigured [16]. Similarly, some researchers propose that -- because systems have different architectural styles, properties of interest, and modification mechanisms -- the control model must be tailored to a specific system [10]. Fortunately, selecting well-known patterns can mitigate this concern through appropriate generalization, thus promoting reuse [11]. While control-centric architecture is surely useful in the context of large systems (for example, virtualized cloud-computing environments), many applications could benefit from a more application-centric architecture methodology that guides them towards achieving dynamic self-optimization.

III. SEEC-AP EXAMPLES

The SEEC-AP case study employed two software architecture patterns that can be composed hierarchically: *divide-and-conquer* and *pipeline* (Fig. 2). These patterns are commonly used for parallel applications that exhibit high data parallelism and/or task parallelism [22].

A small set of abstract base classes form the foundation of SEEC-AP, including: *architecture*, *controller*, *knob*, and *observable*. These are shown in Fig. 3 along with a set of concrete classes used in the prototype implementation. A set of well-defined object relationships (Fig. 4 (a)) enforce composition rules and separation of concerns. For example, the relationships between controller, observable, and knobs map directly to the layers of responsibility shown in Fig. 1.

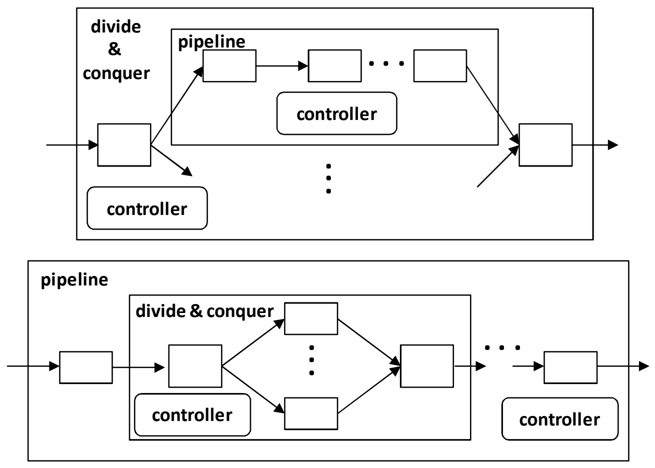


Fig. 2. Composed SEEC-AP patterns

The *basic-unit* class (Fig. 4 (b)) provides a template (the *body*) that the programmer fills in with application-specific code. SEEC-AP patterns collaborate with the basic-unit as follows. The pattern implementation provides structure and control required to sequence work through a set of basic-units, while the body code is invoked by the pattern implementation at the appropriate time, passing in user-defined arguments and eventually receiving user-defined return data. An example set of concrete classes forming an instance of the pipeline pattern are shown in Fig. 5, and for the divide-and-conquer pattern in Fig. 6.

A pipeline consists of two or more stages, where each stage is a basic-unit with an associated body provided by the application developer. At runtime, the pattern implementation emits a pair of heartbeats intended for latency observation before and after execution of each stage's body code. Under the example in Fig. 5 these heartbeats can be observed by a controller using the heartrate monitor API to observe the latency of body code execution. Decisions made within the controller (in this case using a deadbeat control strategy) may ultimately lead to adjusting core frequency on behalf of the application code. Note that in this example the pattern framework provides all implementation except the body which must be provided by the application developer. The only further responsibility the application developer has is to specify the latency goals for the pipeline stages, which can be done through a simple API provided for SEEC-AP pattern configuration.

The example divide-and-conquer SEEC-AP implementation in Fig. **6** includes an additional knob available to the controller. This knob allows the controller to adjust the number of worker threads used by the application. Here the application developer must provide a body implementation that can adjust the division of work amongst workers based upon the number of workers. This simple interface allows the controller to make decisions amongst multiple knobs for optimizing the application.

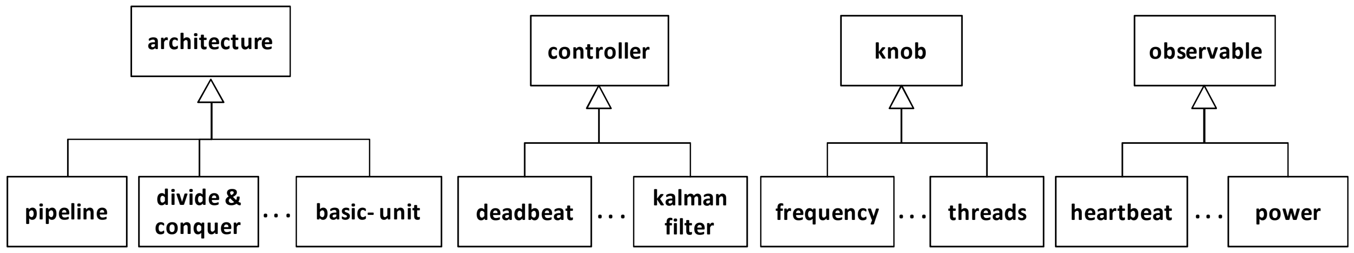


Fig. 3. Class hierarchy for SEEC-AP

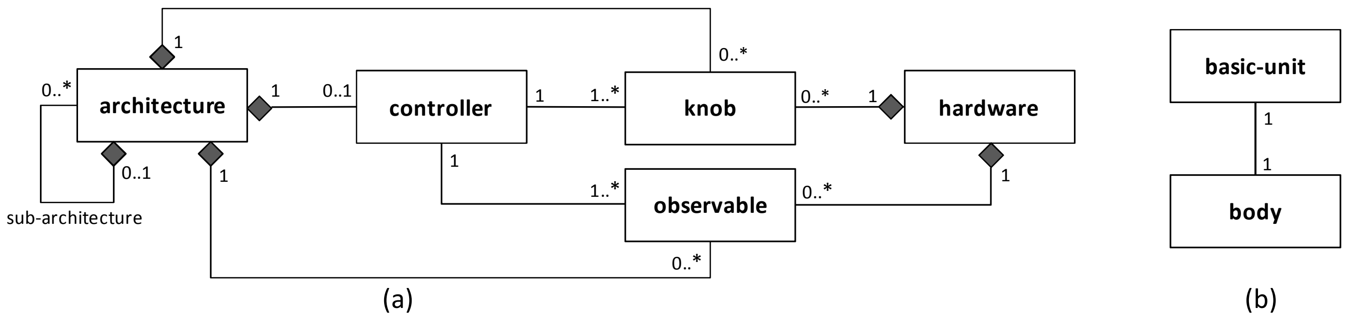


Fig. 4. (a) Object relationships for SEEC-AP, (b) the basic-unit and its body

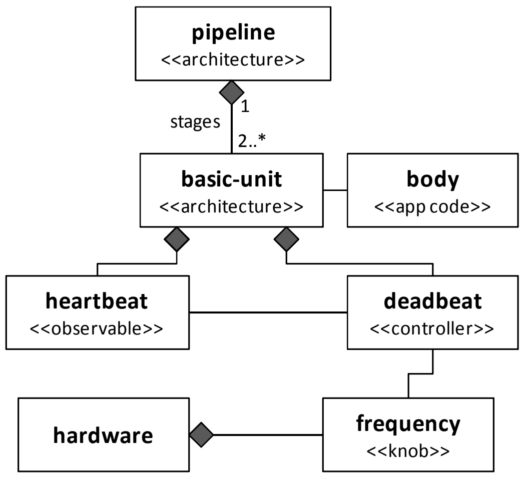


Fig. 5. Objects in SEEC-AP pipeline

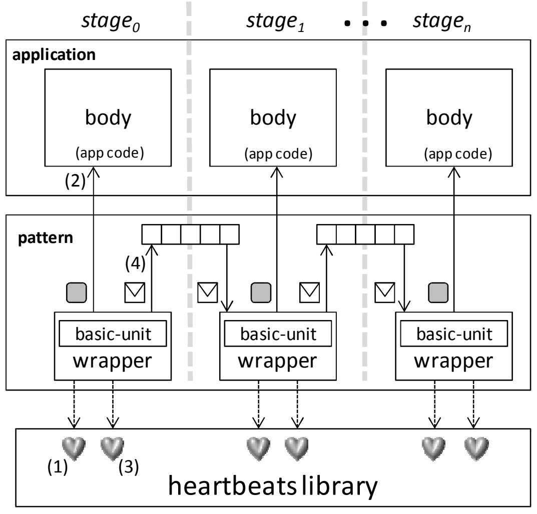


Fig. 7. SEEC-AP pipeline work sequencing

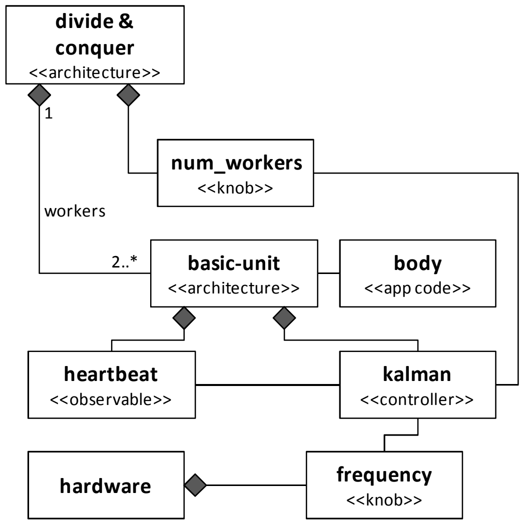


Fig. 6. Objects in SEEC-AP divide-and-conquer

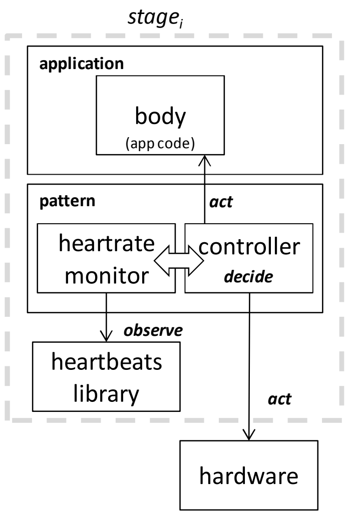


Fig. 8. SEEC-AP controller interaction with basic-unit

The following section provides a deeper look at the prototype SEEC-AP implementation, with emphasis on how interactions with the Heartbeats API and controller are encapsulated within it, as well as how the pattern implementation and application code collaborate to provide application knobs.

IV. Prototype Implementation

The SEEC-AP prototype for the pipeline and divide-and-conquer patterns was created using a simple C-based approach. Fig. 7 shows how the SEEC-AP pipeline sequences work through the application code provided by bodies associated with each stage (processing of data items flows left-to-right in this diagram), while Fig. 8 shows the relations between (i) basic-units (e.g., user-supplied stage implementations in SEEC-AP pipeline, or the worker implementation in SEEC-AP divide-and-conquer), (ii) a per-stage heartrate monitor that observes latency between pairs of heartbeats, and (iii) a per-stage SEEC controller using the heartrate monitor with a deadbeat control algorithm.

Body functions provide an interface to the pattern using the C function signature specified for POSIX threads. The pipeline wraps each body function within a stage wrapper running in a dedicated thread. Additionally, the pipeline provides a mutex synchronized queue for exchanging data between stages. The pipeline implementation performs work sequencing in a loop, as follows (Fig. 7):

1. a heartbeat is emitted by the stage wrapper for the first stage using a *pre-stage$_i$* tag,
2. the body function of the first stage is invoked,
3. once the body function returns (along with body-supplied return data, indicated by the grey rounded rectangles in Fig. 7) a second heartbeat is emitted using a *post-stage$_i$* tag,
4. the return data is wrapped in a generic token and enqueued to the next stage. Intermediate stage wrappers retrieve a token from their input queue, unwrap the user-data, and then follow steps 1-4 above, supplying the data as input to the stage body; the final stage body is expected to consume the data.

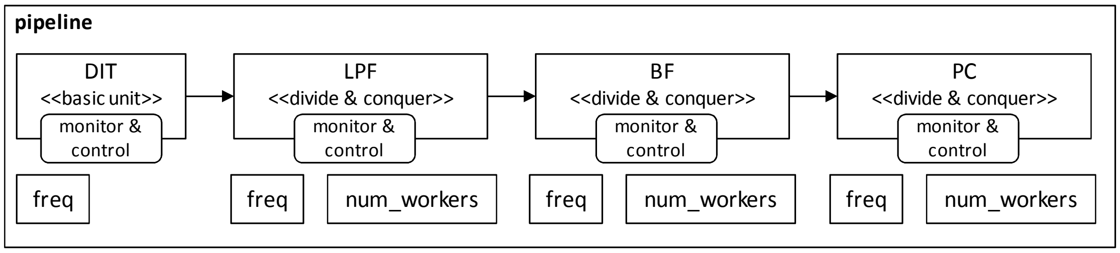


Fig. 9. Mini-SAR pipeline

This sequencing loop proceeds until the first stage returns NULL data. The stage wrapper then terminates the first stage and wraps the NULL data into a token that is enqueued for the next stage. As each stage wrapper encounters NULL data it shuts down its associated body and enqueues the NULL data for the next stage until execution is completed.

Interaction between the controller and basic-units is concurrent with work sequencing in the SEEC-AP patterns. This is illustrated for a single stage in Fig. 8. In the SEEC-AP prototype each controller runs in a dedicated thread within the application. At periodic intervals (based on the heartbeat-related goals supplied by the application) the controller uses the heartrate monitor API to sample heartbeats data. The controller makes decisions based on the application goals for its associated basic-unit (registered when the pattern was initialized). For the pipeline in Fig. 5, the controller can hypothetically increase or decrease frequency for cores executing associated body code.

The controller interaction for the SEEC-AP divide-and-conquer example (Fig. 6) is more involved. This controller may also decide to change the number of workers. Since this decision is made in a controller thread executing concurrently with the threads executing the workers' body code, the SEEC-AP pattern must coordinate carefully. The natural semantics of divide-and-conquer provide guidance to SEEC-AP: adjust the number of workers only outside invocation of the basic-unit. SEEC-AP provides a simple API for these adjustments. The API indicates which divide-and-conquer instance should be updated and whether to increase or decrease the number of workers. The API internally uses a mutex to coordinate this interaction. The mutex is locked during the basic-unit's execution, and unlocked outside basic-unit invocations. When the controller calls the API to adjust workers, the function blocks until it can lock the mutex, releasing the mutex when done. Similarly, before the pattern implementation invokes the basic-unit it will block until it can lock the mutex, again releasing the mutex when done. The benefits of composition become clear when, for example, one of the pipeline stages is composed with an instance of divide-and-conquer. As the pipeline stage repeatedly invokes a composed stage (ultimately invoking the divide-and-conquer pattern), adjustments between stage invocations will provide for thread-safe reconfiguration.

V. CASE STUDIES

The SEEC-AP prototype was evaluated using two streaming benchmarks: synthetic aperture radar benchmark (*mini-SAR*) [23], and the UHPC streaming sensor challenge benchmark (*streaming sensor*) [24]. These benchmarks were chosen because they have high degrees of both task- and data-parallelism, and they are conceived as long-running applications with varying stages of processing, wherein some stages are more compute-intensive than others. Thus, they represent a class of applications with many potential configurations that can benefit from dynamic self-optimization in a variety of deployment environments.

The overall SEEC-AP mapping of mini-SAR is shown in Fig. 9. Four stages are arranged in a SEEC-AP pipeline: data input task (DIT), low pass filter (LPF), beam forming (BF), and pulse compression (PC). Three stages (LPF, BF, and PC) have high data-parallelism, therefore the SEEC-AP divide-and-conquer pattern is composed with the pipeline for these stages. The DIT stage simply uses a SEEC-AP basic-unit. Per-stage deadbeat controllers monitor heartrates and can adjust core frequency for the DIT stage, or core frequency as well as number of workers for all other stages. The streaming sensor benchmark has a similar architecture with 5 stages: data input (*readdata*), image formation (*formimage*), affine transformation (*affine*), coherent change detection (*CCD*), and constant false alarm rate (*CFAR*). The bottleneck stages for streaming sensor (e.g., formimage and affine) utilize the SEEC-AP divide-and-conquer composition, while other stages simply use a basic-unit.
Steps to port these benchmarks to SEEC-AP were as follows: (1) modify functions for each stage processing step into a thread-safe function with a POSIX thread-compliant interface (following processing termination conventions discussed in Section IV), (2) create a data structure for context information to be passed between stages (using the SEEC-AP queueing conventions discussed in Section IV), (3) add functions for adjustments to number of workers for SEEC-AP divide-and-conquer.

The SEEC-AP adaptation API only indicates whether the controller wishes to increase or decrease the number of workers. Choice of increments for number of workers are necessarily application dependent (for example streaming sensor workers must be an integer factor of image dimensions). Thus the adjustment function first calculates the new number of workers, and then updates internal data structures used by workers to indicate each worker's data processing sub-range within the larger dataset.

Experiments were run on a Dell PowerEdge R410 server with two quad-core Intel Xeon E5530 processors and Linux 2.6.26. The processors support seven power states with clock frequencies from 1.596 GHz to 2.394 GHz. The *cpufrequtils* package enables software control of the clock frequency. Hyperthreads were disabled for this paper. Power is measured by a WattsUp device which samples and stores power at 1 second intervals [25]. All benchmark applications run for significantly more than 1 second so the sampling interval should not affect results. The maximum and minimum measured power ranges from 220 watts (at full load) to 80 watts (idle), with a typical idle power consumption of approximately 90 watts. Note also, that hardware was over-provisioned with respect to mini-SAR heartbeat goals, while under-provisioned for streaming sensor goals.

Fig. 10 shows SEEC-AP optimization results for the mini-SAR benchmark. Data was collected for 10000 SAR pulses with a target heartrate latency of 8.33 milliseconds using 50 manual configurations of the application (manually selected combinations of thread counts and core frequencies) and two self-optimizing configurations. The graph shows two manual configurations (*full fury* and *manual@target*) alongside self-optimizing configurations (*HB+freq*, *HB+freq+threads*). *Full fury* achieved the highest heartrate of any manual configuration, while *manual@target* achieved the best combination of target heartrate and lowest power for manual configurations. Each group of bars represents one configuration of the mini-SAR app (with a bar indicating heartrate of each stage plus a bar for the entire application), while the solid curve shows average power for each configuration. The HB+freq configuration allowed frequency changes, while the HB+freq+threads also allowed changes to number of worker threads. The full fury configuration exceeded the target heartrate and also had the highest average power. The manual@target configuration. achieved the target heartrate with improved power. However, variance in per-stage heartrate was still high. Both manual configurations exhibited high power consumption in the DIT stage because DVFS was not employed (this all cores were configured to meet the demands of the most processing intensive stages). Each of the self-optimizing configurations achieved less variance in heartrate across stages and lower power than manual configurations. Lower variance illustrates that SEEC-AP can free up processor resources for other tasks. However, HB+freq+threads achieved slightly less than the target heartrate. This indicates a need for finer grained control in the application or hardware.

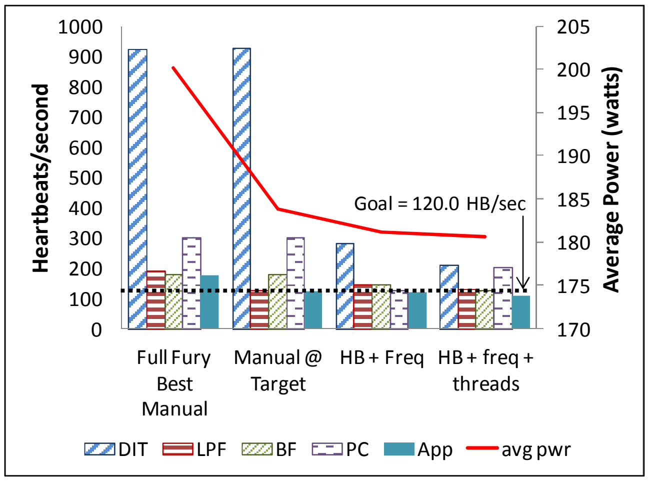


Fig. 10. Mini-SAR self-optimization results

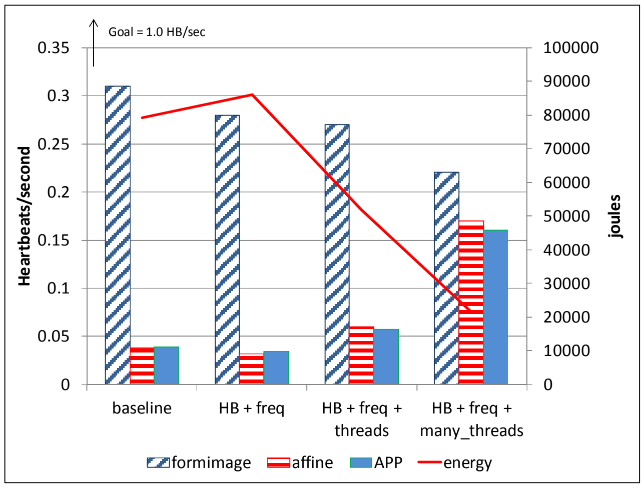
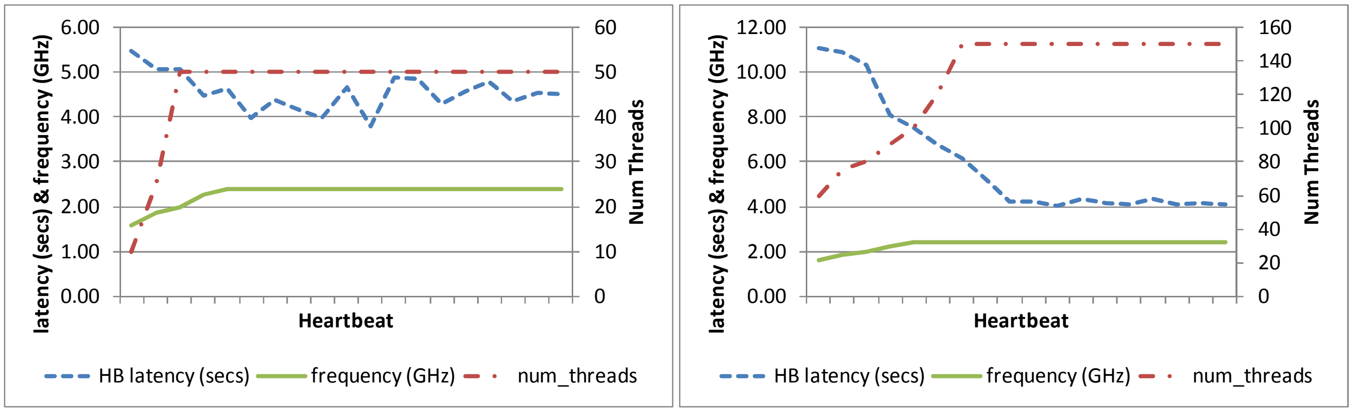


Fig. 11. Streaming sensor self-optimization results

Fig. 11 shows SEEC-AP performance and power optimization results for the streaming sensor benchmark (the formimage and affine stages are extreme bottlenecks). This data was collected by processing twenty 250x250 pixel sensor images. Note that the streaming sensor application is much more compute intensive than mini-SAR, with enough data parallelism to employ 1000's of threads in the formimage and affine stages. However, because this data was collected on an 8 processor host, the evaluation limited baseline, HB+freq, and HB+freq+threads to 20 threads, and allowed HB+freq+many threads to utilize up to 200 threads. Even with the overhead of many threads, performance and power results were improved by self-optimization. Although manual configuration can achieve similar results, the observation is that SEEC-AP found this configuration on behalf of the programmer.



(a) (b)

Fig. 12. Streaming sensor self-optimization profile, (a) *formimage*, (b) *affine*

Mini-SAR	Full Fury	Manual @ Target	HB+Freq	HB+freq+threads
target heartrate/actual	1.48	1.03	1.01	0.91
stage latency mean	397.50	382.25	174.75	165.75
stage latency stddev	355.11	369.00	72.09	45.46
stage latency range	744.00	799.00	156.00	84.00

Streaming Sensor	baseline	hb+freq	HB+Freq+threads	HB+freq+many_threads
target heartrate/actual	0.039	0.034	0.057	0.16
stage latency mean	17.77	6.19	7.05	3.71
stage latency stddev	26.24	8.29	8.35	3.29
range	60.62	19.83	20.44	7.05

Fig. 13. Improvements from SEEC control

Fig. 12 illustrates the profile of core frequency and worker threads for the formimage and affine stages. Note that the controller quickly converges on frequency and threads to meet goals. Other data generated using higher threadcounts on a manycore simulator indicate that both of these stages can benefit from very high thread counts ($>10^3$), although this cannot be represented well on the 8 core workstation used to generate the SEEC-AP data in this paper. Despite this limitation in the experimental setup, SEEC-AP data suggests that the techniques employed are effective at achieving control, while other factors such as ease-of-use are more subjective.

Fig. 13 summarizes improvements in stage latency variation with SEEC. Mini-SAR improved standard deviation between 79% and 89%, while average power reduction is approximately 10%. Streaming sensor standard deviation improvement ranges from 68% to 87%, while total energy reduction ranges from 35% to 73%.

VI. Related Work

Ramirez and Cheng used patterns to reconfigure systems for changes in the environment or for new requirements [11]. The method employs ODA and promotes reuse via a set of patterns to facilitate self-management and self-reconfiguration. Of these, *Adaptation Detector* and *Tradeoff-based* are similar to, but more control-centric than, SEEC-AP.

Self-adaptating architectures using *producer* and *evolver* components was proposed by [18]. This includes concepts of (i) probes used by the evolver to make decisions, (ii) a change interface to reconfigure the producer, and (iii) support for composition of components. However this did not take advantage of the semantics of the producer's software architecture to encapsulate mechanisms and policies for adaptations (as in SEEC-AP).

A UML profile for explicit hierarchical control loops has been defined [15]. This provides value to the control loop implementer by providing warning signals and analysis techniques to identify issues in a design. In contrast, SEEC-AP aids the programmer by encapsulating the control loop into a pre-verified and re-usable framework implementation.

SAFCA defines self-adaptive architectures for *dynamic-thread-creation* (DTC) and *half-sync/half-async* (HS/HA) concurrency architectures [26]. A SAFCA application dynamically switches between these architecture styles based on system load (DTC and HS/HA have different power/performance profiles). SEEC-AP assumes that provisioning of resources will change within a single application architecture, which may be less complex.

VII. Conclusions

This paper demonstrates an application-centric, self-aware architecture framework. Goals include: encapsulating and promoting reuse of self-optimization patterns while separating concerns between the pattern and application implementers.

Although the SEEC-AP prototype implementation encapsulates everything within a single program, ultimately portions of this should migrate to operating systems and runtime libraries.

SEEC provides many sophisticated controllers, including machine learning. SEEC-AP, by virtue of encapsulation of the controller, should make substituting control strategies quite easy, without significant change to the application code.

Future studies should investigate (i) runtime overhead and code size for SEEC-AP, and (ii) effectiveness and ease of using various SEEC controllers. Finally, the object-oriented nature of SEEC-AP would best be implemented as a set of C++ templates to promote better type-safety.

Acknowledgements

This work was funded by the U.S. Government under the DARPA UHPC program. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either express or implied, of the U.S. Government.

References

[1] M. Telgarsky, J. C. Hoe, and J. M. F. Moura, "Spiral: Joint Runtime and Energy Optimization of Linear Transforms," presented at Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on, 2006.

[2] H. Hoffmann, J. Holt, G. Kurian, E. Lau, M. Maggio, J. E. Miller, S. M. Neuman, M. Siningil, Y. Siningil, A. Agarwal, A. P. Chandrakasan, and S. Devadas, "Self-Aware Computing in the Angstrom Processor," presented at To Appear in 2012 Design Automation Conference, 2012.

[3] T. G. Mattson, B. A. Sanders, and B. A. Massingill, *Patterns for Parallel Programming*: Addison Wesley, 2005.

[4] M. D. Santambrogio, H. Hoffmann, J. Eastep, and A. Agarwal, "Enabling technologies for self-aware adaptive systems," presented at Adaptive Hardware and Systems (AHS), 2010 NASA/ESA Conference on, 2010.

[5] H. Hoffman, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic Knobs for Responsive Power-Aware Computing," presented at Architectural Support for Programming Languages and Operating Systems, 2011.

[6] M. Frigo and S. G. Johnson, "FFTW: an adaptive software architecture for the FFT," presented at Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on, 1998.

[7] S. F. Rahman, J. Guo, and Q. Yi, "Automated empirical tuning of scientific codes for performance and power consumption," in *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*. Heraklion, Greece: ACM, 2011.

[8] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated Empirical Optimization of Software and the Atlas Project," *Parallel Computing*, vol. 27, pp. 3-35, 2001.

[9] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Softw. Eng. Notes*, vol. 17, pp. 40-52, 1992.

[10] D. Garlan, S. W. Cheng, A. C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, pp. 46-54, 2004.

[11] A. Ramirez, J. and B. H. C. Cheng, "Design patterns for developing dynamically adaptive systems," in *2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. Cape Town, South Africa: ACM, 2010.

[12] S.-W. Cheng, D. Garlan, and B. Schmerl, "Architecture-based self-adaptation in the presence of multiple objectives," in *2006 international workshop on Self-adaptation and self-managing systems*. Shanghai, China: ACM, 2006.

[13] D. M. Chess, A. Segal, I. Whalley, and S. R. White, "Unity: experiences with a prototype autonomic computing system," presented at Autonomic Computing, 2004. Proceedings. International Conference on, 2004.

[14] S. Dobson, R. Sterritt, P. Nixon, and M. Hinchey, "Fulfilling the Vision of Autonomic Computing," *Computer*, vol. 43, pp. 35-41, 2010.

[15] R. Hebig, H. Giese, and **B.** Becker, "Making control loops explicit when architecting self-adaptive systems," in *Proceedings of the second international workshop on Self-organizing architectures*. Washington, DC, USA: ACM, 2010.

[16] J. Kramer and J. Magee, "Self-Managed Systems: an Architectural Challenge," presented at Future of Software Engineering, 2007. FOSE '07, 2007.

[17] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An architecture-based approach to self-adaptive software," *Intelligent Systems and their Applications, IEEE*, vol. 14, pp. 54-62, 1999.

[18] D. Balasubramaniam, R. Morrison, G. Kirby, K. Mickan, B. Warboys, I. Robertson, B. Snowdon, R. M. Greenwood, and W. Seet, "A software architecture approach for structuring autonomic systems," *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 1-7, 2005.

[19] B. H. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. M. Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Mueller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle, "Software Engineering for Self-Adaptive Systems: A Research Roadmap," in *Software Engineering for Self-Adaptive Systems*, H. C. Betty, Rog, L. rio, G. Holger, I. Paola, and M. Jeff, Eds.: Springer-Verlag, 2009, pp. 1-26.

[20] M. Hussein and H. Gomaa, "An architecture-based dynamic adaptation model and framework for adaptive software systems," presented at Computer Systems and Applications (AICCSA), 2011 9th IEEE/ACS International Conference on, 2011.

[21] D. Weyns, S. Malek, and J. Andersson, "On decentralized self-adaptation: lessons from the trenches and challenges for the future," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. Cape Town, South Africa: ACM, 2010.

[22] C. Bienia and K. Li, "Characteristics of Workloads Using the Pipeline Programming Model," presented at 3rd Workshop on Emerging Applications and Manycore Architectures, 2010.

[23] H. Hoffmann, A. Agarwal, and S. Devadas, "Selecting Spatiotemporal Patterns for Development of Parallel Applications," *Parallel and Distributed Systems, IEEE Transactions on*, vol. PP, pp. 1-1, 2011.

[24] D. P. Campbell, D. A. Cook, and B. P. Mulvaney, "A Streaming Sensor Challenge Problem for Ubiquitous High Performance Computing," presented at 15th Annual Workshop on High Performance Embedded Computing, 2011.

[25] Watts Up, "Watts Up Power Meter," 2012.

[26] Z. Xu and L. Chung-Horng, "Improving Software Performance and Reliability with an Architecture-Based Self-Adaptive Framework," presented at Computer Software and Applications Conference (COMPSAC), 2010 IEEE 34th Annual, 2010.