Program Phase Detection and Exploitation

Chen Ding, Sandhya Dwarkadas, Michael C. Huang, Kai Shen
Department of Computer Science
University of Rochester
Rochester, NY
{cding,sandhya,mihuang,kshen}@cs.rochester.edu

John B. Carter
Department of Computer Science
University of Utah
Salt Lake City, Utah
retrac@cs.utah.edu

Abstract— **Studies of application behavior reveal the nested repetition of large and small program *phases*, with significant variation among phases in such characteristics as memory reference patterns, memory and energy usage, I/O activity, and occupancy of micro-architectural resources. In this project, we study theories and techniques for reliably predicting and exploiting phased behavior, so an advanced execution environment may allocate resources in a way that better matches program needs, or to transform programs so that their needs better match the available resources. In this paper, we present the basic components of the study and report the progress in the past half year.**

I. Introduction

Despite technology advances, applications such as scientific simulation, weather forecasting, bioinformatics, and data mining continue to demand more computational, memory, and I/O resources than are currently available. The need for scalable and high-performance computing continues to grow, requiring that applications, and correspondingly, the underlying architecture, are finely tuned to match each other's needs.

During the course of execution, programs demonstrate diverse behaviors, which can be characterized as behavior *phases*. Detecting these phases and anticipating the behaviors within them is a crucial prerequisite for static and/or run-time adaptations that improve performance or other aspects of execution. Adaptation techniques may include hardware reconfiguration, data reorganization and redistribution, changes in scheduling and task assignment, and modifications to I/O management.

We have used phases to predict [9] and evaluate [15] performance, to drive the control of adaptive hardware (primarily at the micro-architectural level) [3], [4], [5], [8], [11], [12], [16], and to modify the strategies used for scheduling and load balancing [14], [20], [21], I/O prefetching and caching [18], [19], and data distribution and coherence [1], [2], [6], [24].

Based on our past experience, we envision an advanced execution system along the lines of Figure 1. Applications are instrumented based on static analysis and small-scale profiling, to include "hooks" at likely phase boundaries, and to embody initial estimates of phase behavior. Full-scale runs then incorporate lightweight run-time monitoring to collect statistics on input characteristics and dynamic resource usage, under partial control of the instrumentation hooks. Feedback from this run-time monitoring is then used to refine the instrumentation and build an application profile, for use in future runs. In steady state, the instrumentation and run-time monitor together direct the application of adaptation mechanisms that exploit application phases.

The following sections describe in detail our recent research in phase detection and exploitation at different levels of an execution environment.

1-4244-0054-6/06/$20.00

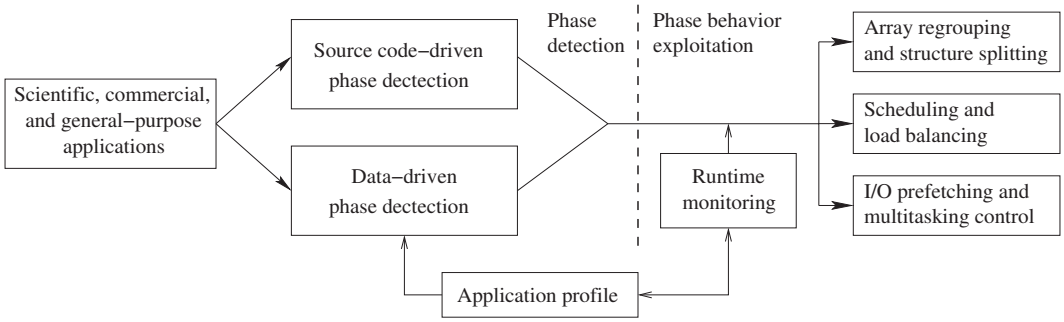

Fig. 1. Overview of research issues.

II. Phase Detection

Informally, a phase is a period of execution whose characteristics are qualitatively different from those of the neighboring periods. A phase detection algorithm must detect events in the execution of an application that signal the transition between phases *and* can accurately forecast behavior in the upcoming phase. The events may be from the instruction stream (*e.g.*, a certain code section is reached), the data stream (*e.g.*, certain data elements have been accessed, or the data access pattern has changed), an asynchronous external event (*e.g.*, the arrival of an incoming message), or a combination of these.

Unfortunately, phase boundaries are not always clear and unambiguous. In particular, they may depend on the behavior being measured, which in turn depends on the optimization technique one hopes to apply. Phases also tend to nest, reflecting the hierarchical structure of many algorithms. At higher levels of the hierarchy the behavior inside a phase may not be uniform.

One can subdivide the design space of detection algorithms based on whether a "top-down" or a "bottom-up" approach is used, and whether the detection is performed online or offline (Figure 2). In a top-down approach, execution is divided into *candidate* phases based on the high-level structure of the source code: the beginnings of long-running subroutines and loops mark the potential boundaries between phases. In a bottom-up approach, one starts with the behavior metrics observed during execution, and looks for recurring patterns and changes.

Both approaches can be carried out entirely online, as the program is executing, or partially offline, using feedback from prior executions or from small-scale profiling or sampled simulation. The top-down approach typically requires modest compile-time instrumentation to insert instructions at candidate phase boundaries. The bottom-up approach can be performed entirely on-line, with unmodified program binaries, but is likely to be strengthened considerably by going back to the source code to correlate observed phase transitions with one or a group of static instructions.

Offline phase analysis has two important advantages. First, it inserts phase markers into a program. At run time, a phase marker is "active" rather than reactive. It allows immediate recognition of a phase change and precise monitoring of phase behavior at run time. In comparison, purely online detection detects a phase change only after the change is happened. Second, repetition patterns are usually associated with repetition in the code [15]. The static code associated with a phase also provides a natural link between different executions of the same program. Purely online phase detection mechanisms have no natural way to pass information from one execution of a program to the next. Our past work in architectural adaptation [17], [16] suggests that information from prior executions can help us make significantly better decisions. We thus propose to focus on the profile-driven (partially offline) half of the design space.

Next we describe two profile-driven strategies. The first is bottom-up: it monitors a run-time metric (data reference locality) and looks for patterns and changes. The second is top-down: it identifies candidate phases (long-running subroutines and loop nests) and looks for repetitions and consistency. Both insert phase markers into the source or the object code.

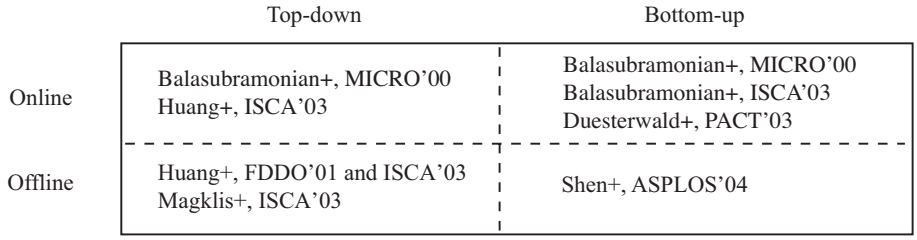

Fig. 2. Design space of phase detection and our recent papers in major conferences.

A. Data and Input-Driven Bottom-Up Analysis

Many programs have recurring locality phases. For example, a simulation program may test the dynamics of some complex physical system. The computation sweeps through a mesh structure repeatedly in many time steps. The cache behavior of each time step should be similar because the majority of the data access is the same despite local variations in control flow. Given a different input, for example, another physical system, the previously identified phase of program execution will again have similar locality in the new execution, although the phase locality may be very different from the previous one. Phase behavior of this sort is common in structural, mechanical, molecular, and other scientific and commercial simulations.

We have introduced a 3-step phase detection and behavior prediction method based on reuse distance [23]. The first step uses small-scale profiling runs to analyze data locality. By examining data reuses of varying lengths, the analysis can "zoom in" and "zoom out" over long execution traces. It detects locality phases using a statistical method *variable-distance sampling*, *wavelet filtering*, and *optimal phase partitioning*. The second step then analyzes the instruction trace and correlates the phase boundaries with static instructions in the application binary. The third step uses grammar compression to identify phase hierarchies and then inserts marker instructions into the application via binary rewriting. (No access to source code is required.) During execution, the interval between marker instructions constitutes a dynamic phase instance. The program monitors the behavior of the first few dynamic instances and uses it to predict the behavior of subsequent instances. Because analysis is bottom up, it considers all instructions in the program binary as potential phase-change points and can handle programs where the loop and procedure structures are obfuscated by an optimizing compiler.

GCC is the GNU C compiler included in SPEC2K benchmark set. It represents a class of dynamic programs whose behavior depends strongly on their input. The behavior varies not only from one input to another but also across sections of the same input. Other examples include interpreters, compression and transcoding utilities, databases, and web servers. These applications share the common feature that they provide some sort of *service*: they accept, or can be configured to accept, a sequence of requests, each of which is processed more-or-less independently of the others. Because they are so heavily dependent on their input, service-oriented applications display much less regular behavior than does the typical scientific program.

We have developed a new technique called *active profiling*, in which we provide a service-oriented application with an artificially regular input and use the bottom-up phase analysis to identify behavior phases and mark them in the program executable. Active profiling differs from traditional profiling in that its input is specially designed to expose desired behavior in the program. Our analysis finds the compilation loop and its inner phases and inserts eight markers in the program that includes 120 files and 222182 lines of C code at the source level. Given the complexity of the code, manual phase marking would be extremely difficult for someone who does not know the program well. Even for an expert in *GCC*, it may not be easy to identify sub-phases that occupies a significant portion of the execution time.

Figure 3 shows the phase behavior of *GCC*. The left-hand graph shows the IPC (instruction per

cycle) curve for the regular input. The IPC is measured on IBM using hardware counters. Each point on the curve is the IPC of 10 ms of the execution. Based on the regular behavior, our phase analysis marks the phases in the program. The right-hand graph shows the IPC curve for a reference input, 166.i. The phase and sub-phase instances are separated by solid and dotted lines respectively. The presence of a regular pattern is not obvious from visual inspection until it is divided into phases. Although phase instances have different widths and heights, they show a similar signal shape—with two high peaks in the middle and a declining tail. Testing on other inputs shows a similar result, which means that *GCC* has a consistent program pattern—the same complex compilation stages are performed on each function in each input file. The phase and sub-phase markers accurately capture the variation and repetition of program behavior, even when the shape of the curve is not exactly identical from function to function or from input to input. The phase marking is done off-line and requires no on-line measurement.

Results indicate that data-driven locality phase prediction can be very accurate, even when profiling on very small inputs and then testing on much longer runs. In addition to *Tomcatv* and *GCC*, our tests were conducted on a wide range of applications [23], [22], including a fast Fourier transformation (*FFT*); a finite-element solver (*Swim*); a partial differential equation solver (*AppLU*); two simulation programs, *MolDyn* and *Mesh*; and four service-oriented programs, a UNIX compression utility (*Compress*), a lisp interpreter *LI*, a natural language parser *Parser*, and a object-oriented database *Vortex*.

In our experiments phase size ranged from 2.2 to 98 million instructions in the profiling runs, and from 33 million to 1.7 billion in the production runs. The production runs had up to 150 times more phases and 1000 times longer execution than the profiling runs. Across most test programs, the prediction coverage and accuracy are over 90%.

We also compared locality phase prediction with manual phase marking. We hand-analyzed each program and inserted phase markers based on our reading of the code and its documentation. When a program executes, both automatically and manually inserted markers give a sequence of phase predictions. We measure the overlap between the two sequences using the standard notions of *precision* and *recall*. We omit per-application results due to lack of space. Overall, the average recall is 96% for profiling and 99% for production runs, showing that automatic marker insertion catches all manually marked phase transitions. Precision is also high for the majority of the programs, but low in a few cases because, as we found out through manual inspection of the code, the automatic analysis is more thorough, and catches cases that the programmer may miss. Four of the test programs were grid, mesh, or N-body simulations with time steps. In all cases, the analysis identified the time step loop as the largest composite phase.

We will extend this framework to more complex and dynamic programs and to study phase detection in parallel programs. The PSIMUL tool at IBM showed recurring memory-access patterns among processes of parallel scientific programs [7], indicating that we may apply our bottom-up framework to identify repeating locality patterns and mark phases in the program.

B. Top-Down Phase Detection

In top-down phase detection, we begin by identifying long-running subroutines and loop nests, each of which is then tagged with marker instructions in the program executable. At run time the marker instructions trigger the execution of library code that tracks the metrics of interest, identifying points at which changes suggest the need for dynamic adaptation.

We follow a systematic methodology to identify major code structures: long-running subroutines and long-running loop nests within them. To avoid instrumenting small subroutines and loops (which could result in noticeable and unnecessary overhead in production runs), we begin with a profiling run that instruments all subroutines and loops, constructs a dynamic call tree, and measures the aggregate duration of each node in the tree [16]. We then traverse the tree from the leaves to the root, identifying as “long-running” each node whose aggregate duration, excluding that of long-running children, exceeds a given threshold. For production runs, we then instrument only the long-running

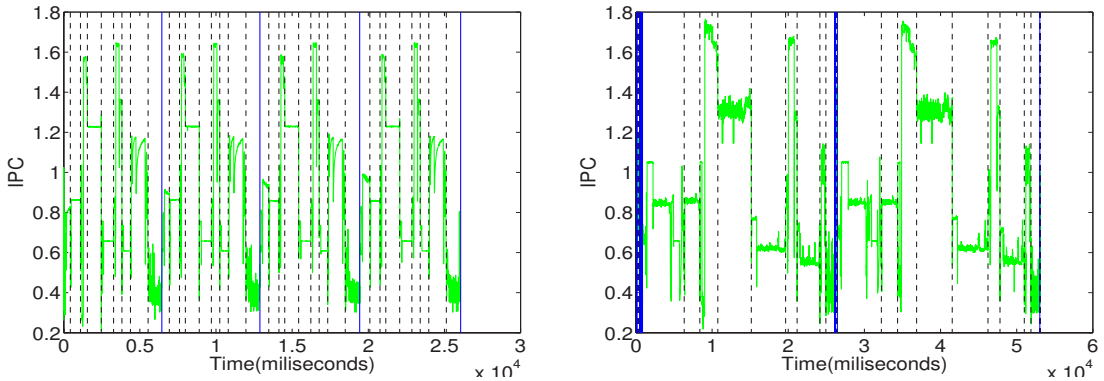

Fig. 3. Phase behavior of *GCC* from SPEC2K. Instances of the outer phase are separated by solid (vertical) lines and inner phases by dotted lines. The left-hand figure shows the training input with regular behavior. The right-hand figure shows a reference input, 166.i. The phase markers accurately identify the phase behavior even though the length of the (sub)-phase instances is input dependent.

nodes. This methodology ensures that, with high probability, each instrumented code section (call tree node) runs long enough to justify the overhead of checking behavioral metrics, yet briefly enough to capture every phase transition worth exploiting. Moreover, every application phase, at every hierarchical level of granularity, begins with an instrumented code section. Experiments with a wide range of applications indicate very high correlation in coverage and accuracy between profiling and production runs.

Phase behavior is often strongly tied to source code structure, as evident from data in Figure 4, drawn from our recent work [15]. This figure summarizes the COV (coefficient of variation: standard deviation divided by the mean) among all dynamic instances of each code section (long-running subroutine or loop nest), for a variety of behavioral metrics, on both integer and floating-point applications. In general the COV values are very low. Most floating-point applications, in particular, have an average COV of only a few percent, indicating that the code sections have very small behavior variations.

Each execution phase follows a unique marker (or marker augmented with calling history) that serves as an ID to the phase. At runtime, each marker will transfer control to a library that implements dynamic behavior monitoring mechanisms and phase-based exploitation algorithms. This mechanism lends itself well to behavior prediction based on history. Behavior of past instances (in the same or an earlier program execution) of the same code can be recorded and used to predict its future behavior. Based on extra behavior characterizations, each different exploitation technique may choose to react to a different subset of the phase markers as the behavior aspect of interest differs from one optimization to another and not every aspect will change significantly across these markers.

To monitor execution behavior, we use commonly available hardware counters (for dispatched and committed instructions, instruction mix, cache misses, branch mis-predictions, queue and functional unit occupancy rates, etc.), as well as software instrumentation (for higher-level operations like synchronization, loop counts, context switching, I/O operations, message delivery, etc.)

III. On-going Projects

We describe three current projects that improve the execution environment at chip multi-processor, program, virtual machine, and uni-processor levels.

A. Compatible Phase Co-Scheduling on Multi-Threaded CMP

The industry is rapidly moving towards the adoption of Chip Multi-Processors (CMPs) of Simultaneous Multi-Threaded (SMT) cores for general purpose systems. The most prominent use of such processors, at least in the near term, will be as

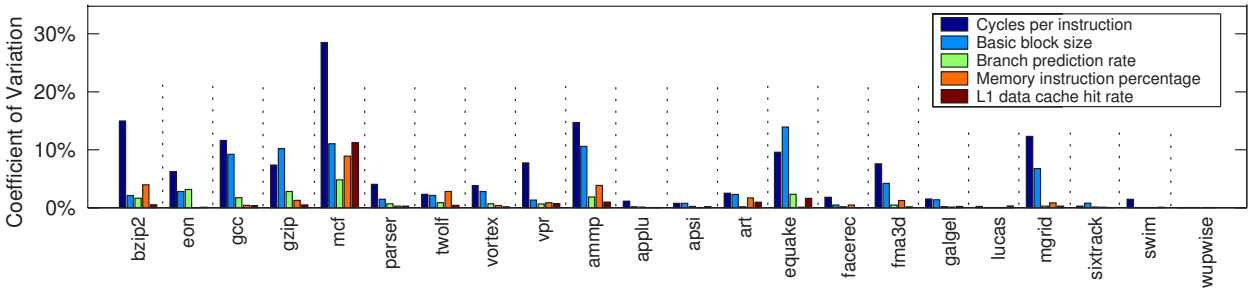

Fig. 4. Average variability of different dynamic code section instances for SPEC 2000 applications.

job servers running multiple independent threads on the different contexts of the various SMT cores. In such an environment, the co-scheduling of phases from different threads plays a significant role in the overall throughput. Less throughput is achieved when phases from different threads that conflict for particular hardware resources are scheduled together, compared with the situation where *compatible phases* are co-scheduled on the same SMT core. Achieving the latter requires precise per-phase hardware statistics that the scheduler can use to rapidly identify possible incompatibilities among phases of different threads, thereby avoiding the potentially high performance cost of inter-phase contention.

We have developed phase co-scheduling policies for a dual-core CMP of dual-threaded SMT processors [10]. We explore a number of approaches and find that the use of ready and in-flight instruction metrics permits effective co-scheduling of compatible phases among the four contexts. This approach significantly outperforms the worst static grouping of threads, and very closely matches the best static grouping, even outperforming it by as much as 7%.

B. Adaptive Memory Management

With behavior phases, the adaptive heap sizing becomes feasible for utility programs for three reasons. First, a phase represents a memory usage cycle. Garbage collection is best applied at the phase boundaries, because all temporary objects will be dead and ready for collection. Second, when a utility program takes many independent requests, the average execution time per phase instance tends to be consistent because of the *Law of Large Numbers*. By measuring the average speed at run time, an adaptive method can estimate the effect of different heap sizes on program performance. Finally, the granularity of a phase is much larger than the granularity of a statement or a procedure, so an adaptive scheme can monitor execution at the phase boundaries without minimal observable overhead.

We have implemented and tested several schemes. Figure 5 shows the performance comparison for SPEC PseudoJBB, a e-commerce benchmark that constructs several databases and performs a series of queries. We test three garbage collectors—mark-sweep (MarkSweep), copy mark-sweep (CopyMS), and generational copy mark-sweep(GenCopy). We limit the physical memory size to be 128MB.

The fixed schemes preset the heap size as shown by x-axis as a parameter to the virtual machine before the program runs. As shown by Figure 5, the performance for most heap sizes is factor of two to five worse than the performance for the best heap size, which is the global minima of the curve. The "sweep spot" differs for different garbage collections—60MB for MarkSweep, 105MB for CopyMS, and 120MB for GenMS. The goal of the adaptive schemes is to automatically find the best heap size. As shown by dotted lines, the program experiments with a range of heap sizes. The performance is nearly optimal for CopyMS and 10% to 20% off for the other two garbage collectors in part because of the overhead of monitoring and the cost of trying different heap sizes.

This work is done in collaboration with Chengliang Zhang, Kirk Kelsey, and Xipeng Shen at Rochester and Matthew Hertz at Canisius College.

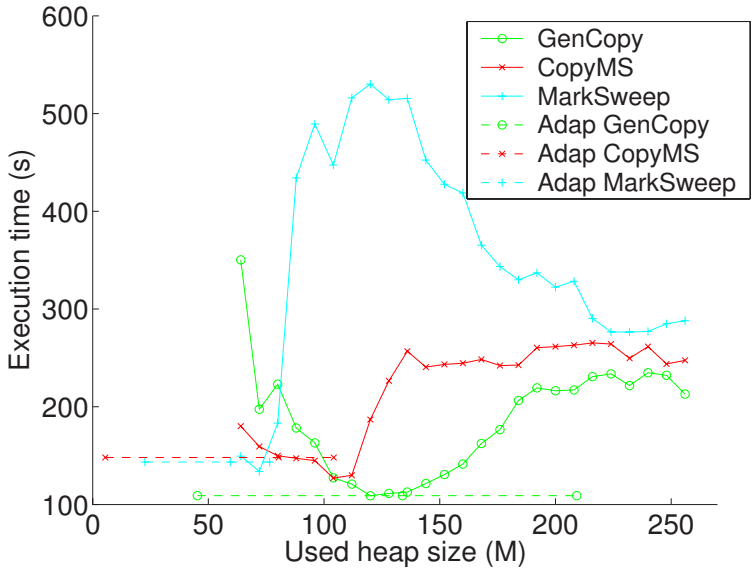

Fig. 5. The performance of SPEC Pseudo JBB using adaptive heap sizing (the leveled lines) and fixed heap sizes in the Jike's RVM using three of its garbage collectors, mark-sweep (MS), copy mark-sweep (CopyMS), and generational copy mark-sweep(GenCopy)

C. Software-Hardware Cooperative Memory Disambiguation

We have examined several microarchitectural performance bottlenecks, especially for high-performance numerical applications. We found that the memory disambiguation mechanism that tracks the out-of-order execution of memory operations is often limited in its capacity in handling a large amount of in-flight operations. This in turn limits the hardware's capability of exploiting the instruction-level parallelism of far-apart instructions. We are exploring a range of solutions to anticipate and circumvent these performance bottlenecks. One angle we looked at is to use binary analysis to identify load instructions that under the microarchitecture's practical constraints will not result in an execution order that violates program semantics regardless of its execution timing. These instructions are then handled specially to yield precious microarchitectural tracking resources at run-time to reduce stalling. Our findings show that a significant amount of load instructions can bypass the hardware, not only saving energy but also relieving pressure on an important resource and therefore improving performance [13].

Our simulation results suggest that programs are often slowed down at critical junctions for extended periods of time. These slowdowns are often triggered by various long-latency stalls. Our goal is to develop a systematic approach to accurately predict these stalls and then address them either through counter measures or by allowing the hardware resources to be used in independent tasks. Our near-term plan is to investigate a software-hardware cooperative mechanism that allows a relatively inexpensive diagnostic/scouting thread to be spawned and executed ahead of the real computation.

IV. Summary

In the paper, we have presented the design and the basic research issues in phase detection and exploitation in an advanced execution environment. Significant empirical evidence show the effectiveness of bottom-up and top-down phase detection schemes. Based on these techniques, we have obtained encouraging results on dynamic memory management, co-scheduling on multi-threaded chip multi-processors, and software-hardware cooperative memory disambiguation. At least three papers [10], [13], [25] will appear in 2006 in parallel computing, computer architecture, and programming language conferences and have explicitly acknowledged the support from National Science Foundation (Contract No. CNS-0509270). The sup-

port comes from a program directed by Frederica Darema.

REFERENCES

[1] C. Amza, A. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM protocols that adapt between single writer and multiple writer. In *Proc. of the 3rd Intl. Symp. on High Performance Computer Architecture*, pages 261–271, San Antonio, TX, February 1997.

[2] C. Amza, A. L. Cox, S. Dwarkadas, L.-J. Jin, K. Rajamani, and W. Zwaenepoel. Adaptive protocols for software distributed shared memory. In *Proc. of the IEEE*, March 1999.

[3] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures. In *International Symposium on Microarchitecture*, pages 245–257, Monterey, California, Dec. 2000.

[4] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. A Dynamically Tunable Memory Hierarchy. *IEEE Transactions on Computers*, 52(10):1243–1258, Oct. 2003.

[5] D. Chaver, L. Pinuel, M. Prieto, F. Tirado, and M. Huang. Branch Prediction on Demand: an Energy-Efficient Solution. In *International Symposium on Low-Power Electronics and Design*, Seoul, Korea, Aug. 2003.

[6] D. Chen, C. Tang, X. Chen, S. Dwarkadas, and M. L. Scott. Multi-level shared state for distributed systems. In *Proc. of the 2002 Intl. Conf. on Parallel Processing*, pages 131–140, Vancouver, BC, Canada, August 2002.

[7] F. Darema, G. F. Pfister, and K. So. Memory access patterns of parallel scientific programs. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1987.

[8] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M. Scott. Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 141–152, Charlottesville, Virginia, Sept. 2002.

[9] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and Predicting Program Behavior and its Variability. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 220–231, New Orleans, Louisiana, Sept. 2003.

[10] A. El-Moursy, R. Garg, D. H. Albonesi, and S. Dwarkadas. Compatible phase co-scheduling on a cmp of multi-threaded processors. In *Proceedings of 2006 International Parallel and Distribute Processing Symposium (IPDPS)*, April 2006.

[11] M. Huang, D. Chaver, L. Pinuel, M. Prieto, and F. Tirado. Customizing the Branch Predictor to Reduce Complexity and Energy Consumption. *IEEE Micro*, 23(5):12–25, Sept. 2003.

[12] M. Huang, J. Renau, and J. Torrellas. Positional Adaptation of Processors: Application to Energy Reduction. In *International Symposium on Computer Architecture*, pages 157–168, San Diego, California, June 2003.

[13] R. Huang, A. Garg, and M. Huang. Software-Hardware Cooperative Memory Disambiguation. In *International Symposium on High-Performance Computer Architecture*, Austin, Texas, Feburary 2006.

[14] S. Ioannidis, U. Rencuzogullari, R. Stets, and S. Dwarkadas. CRAUL: Compiler and run-time integration for adaptation under load. *Journal of Scientific Programming*, pages 261–273, August 1999.

[15] W. Liu and M. Huang. EXPERT: Expedited Simulation Exploiting Program Behavior Repetition. In *International Conference on Supercomputing*, St. Malo, France, June–July 2004.

[16] G. Magklis, M. Scott, G. Semeraro, D. Albonesi, and S. Dropsho. Profile-based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Microprocessor. In *International Symposium on Computer Architecture*, pages 14–25, San Diego, California, June 2003.

[17] G. Magklis, G. Semeraro, D. H. Albonesi, S. Dropsho, S. Dwarkadas, and M. L. Scott. Dynamic frequency and voltage scaling for a multiple clock domain microprocessor. *IEEE Micro*, 23(6):62–68, November–December 2003.

[18] A. E. Papathanasiou and M. L. Scott. Energy efficiency through burstiness. In *Proc. of the 5th IEEE Workshop on Mobile Computing Systems and Applications*, Monterey, CA, October 2003.

[19] A. E. Papathanasiou and M. L. Scott. Energy efficient prefetching and caching. In *Proc. of the USENIX 2004 Technical Conf.*, Boston, MA, June-July 2004.

[20] U. Rencuzogullari and S. Dwarkadas. Dynamic adaptation to available resources for parallel computing in an autonomous network of workstations. In *8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2001.

[21] U. Rencuzogullari and S. Dwarkadas. A technique for adaptation to available resources on clusters independent of synchronization methods used. In *International Conference on Parallel Processing*, Aug. 2002.

[22] X. Shen, C. Ding, S. Dwarkadas, and M. L. Scott. Characterizing phases in service-oriented applications. Technical Report TR 848, Department of Computer Science, University of Rochester, November 2004.

[23] X. Shen, Y. Zhong, and C. Ding. Locality Phase Prediction. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, Massachusetts, Oct. 2004.

[24] R. Stets, S. Dwarkadas, L. I. Kontothanassis, U. Rencuzogullari, and M. L. Scott. The effect of network total order, broadcast, and remote-write capability on network-based shared memory computing. In *Proc. of the 6th Intl. Symp. on High Performance Computer Architecture*, Toulouse, France, January 2000.

[25] C. Zhang, C. Ding, M. Ogihara, Y. Zhong, and Y. Wu. A hierarchical model of data locality. In *Proceedings of ACM Symposium on Principles of Programming Languages*, Charleston, SC, January 2006.